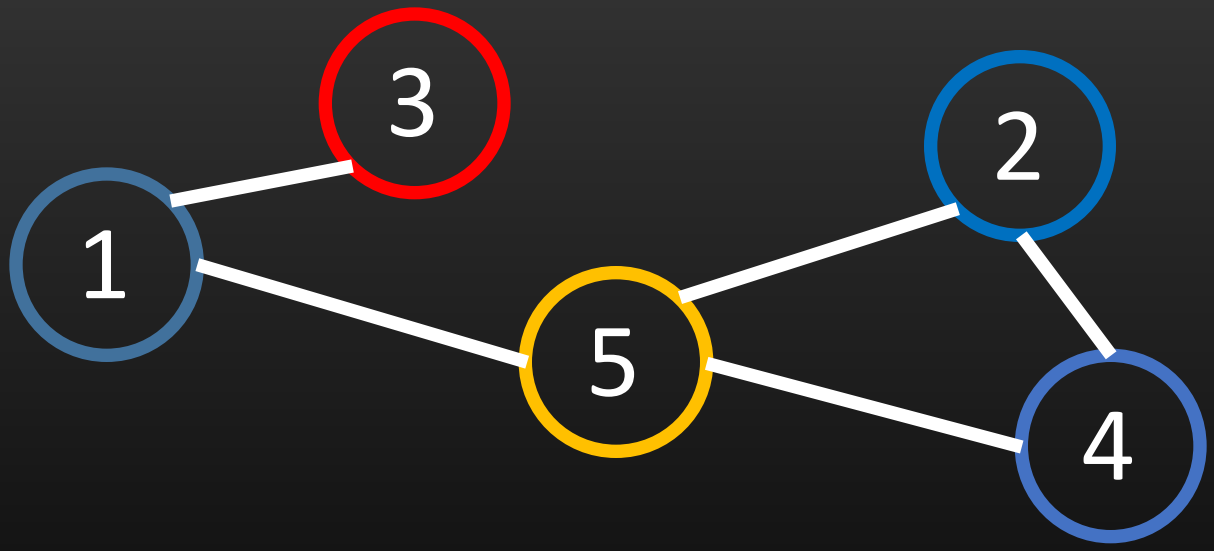
GRAPH I

Jason

Graph

- Consists of a set of vertices and a set of edges
- Each edge connects a pair of vertices



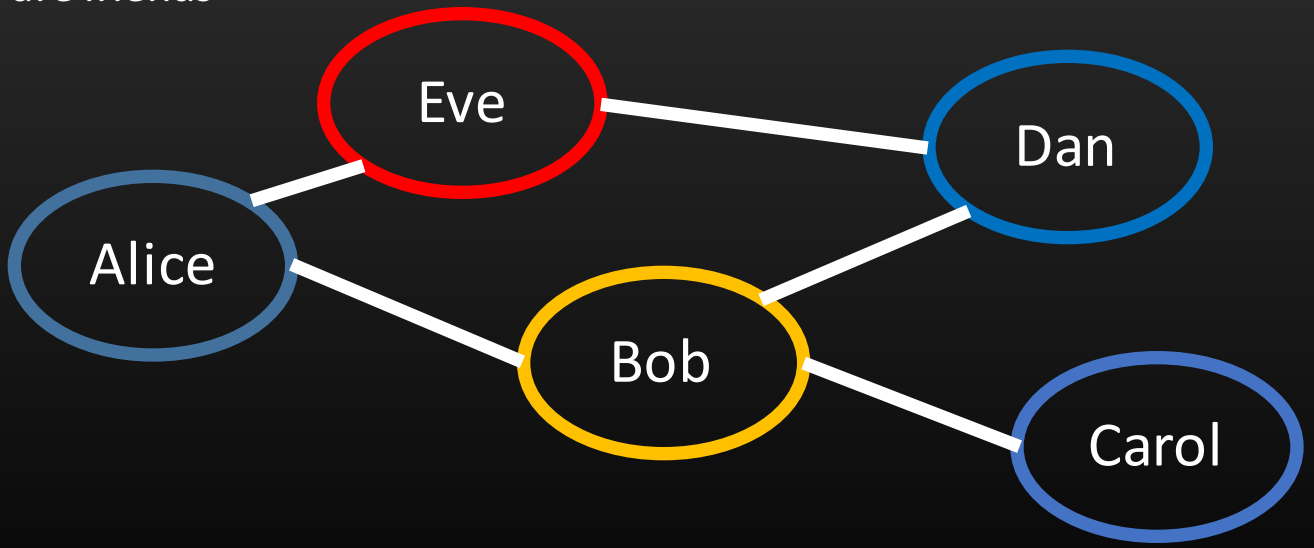
- Mathematically, we often write G = (V,E)
  - V: set of vertices, so |V| = number of vertices
  - E: set of edges, so |E| = number of edges

Usage

- To present the relationships between different objects/elements in a mathematical way
- Examples:
  - Social Networks
  - Maps
  - Grids
  - States

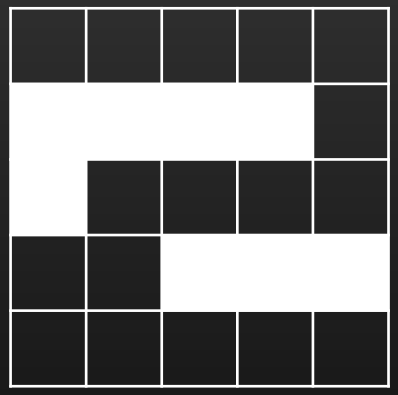
Usage - Example

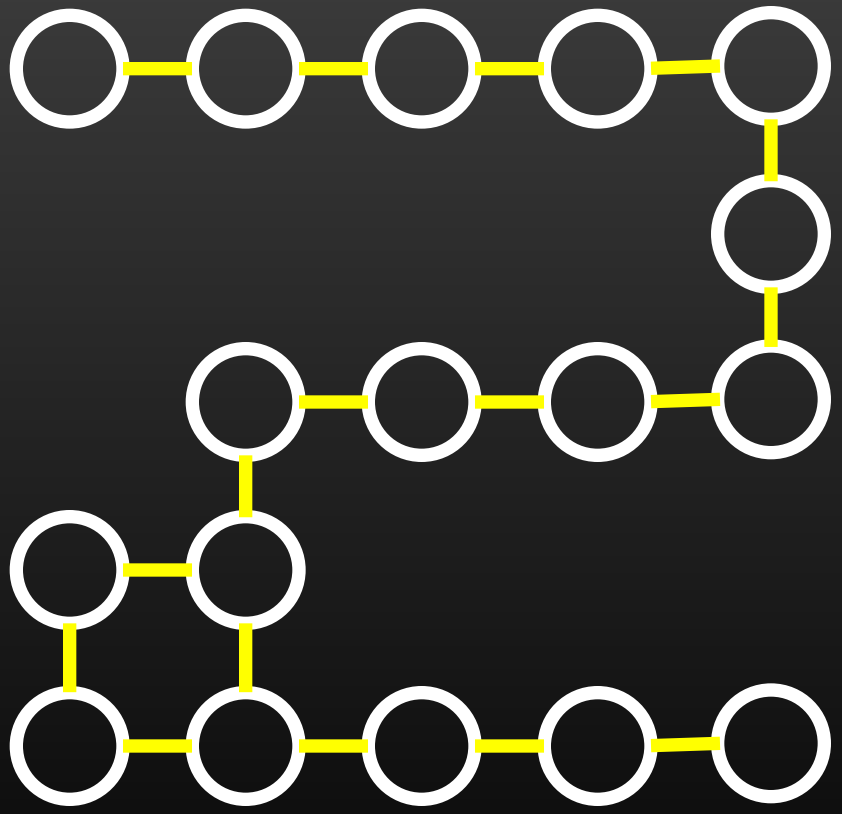
- Social Networks
  - Alice and Bob are friends
  - Bob and Carol are friends
  - Bob and Dan are friends
  - Dan and Eve are friends
  - Alice and Eve are friends



Usage - Example

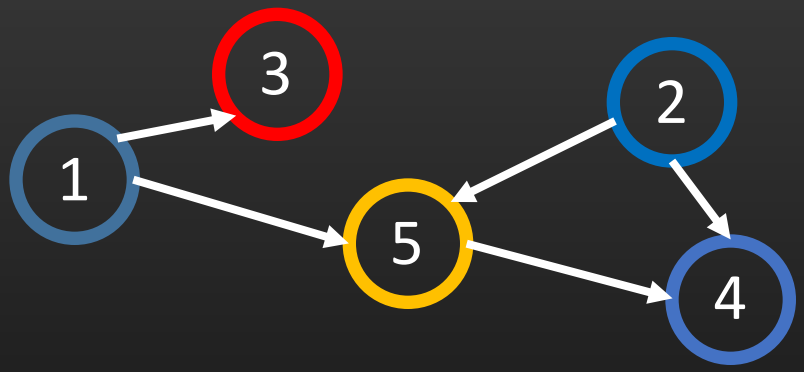
- Maze



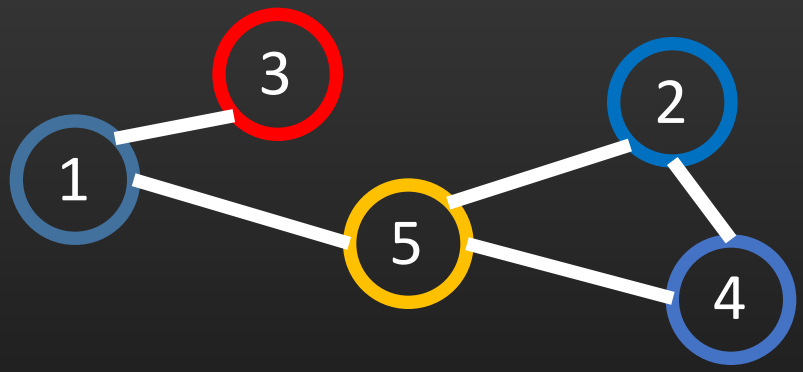


Graph

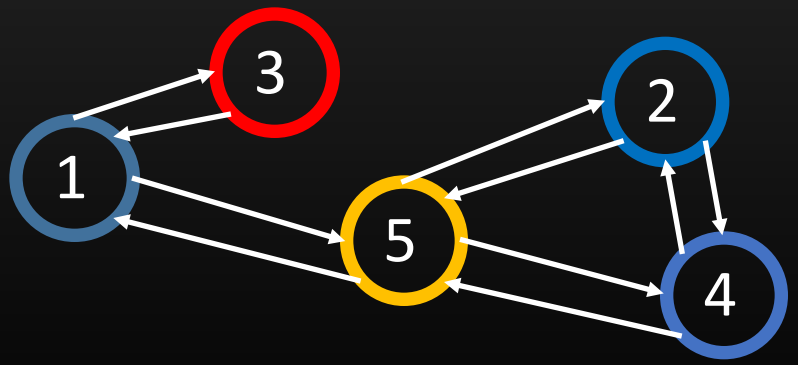
Directed graph:



Undirected graph:

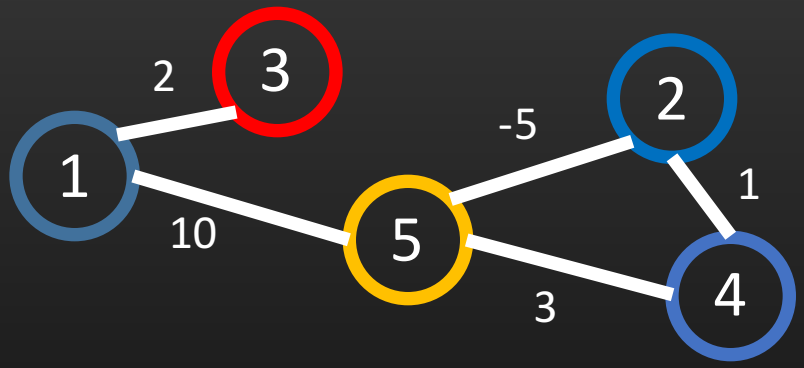


- You may use two directed edges to represent an undirected edge

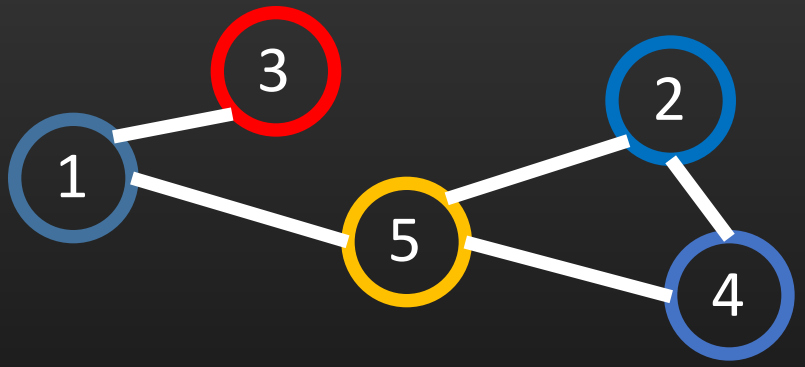


Graph

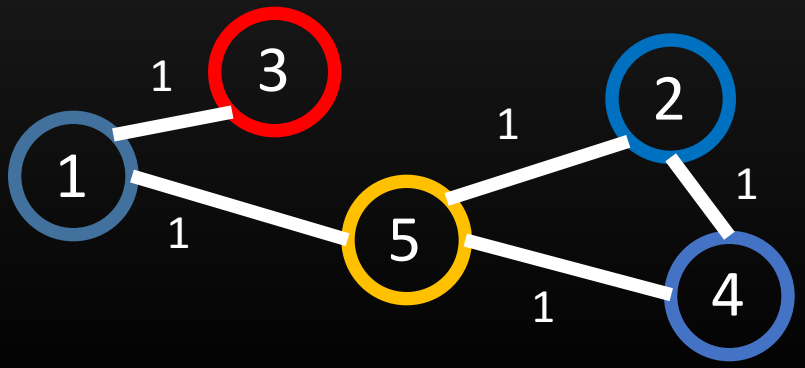
Weighted graph:



Unweighted graph:



- You may treat unweighted edges to be weighted edges of equal weights

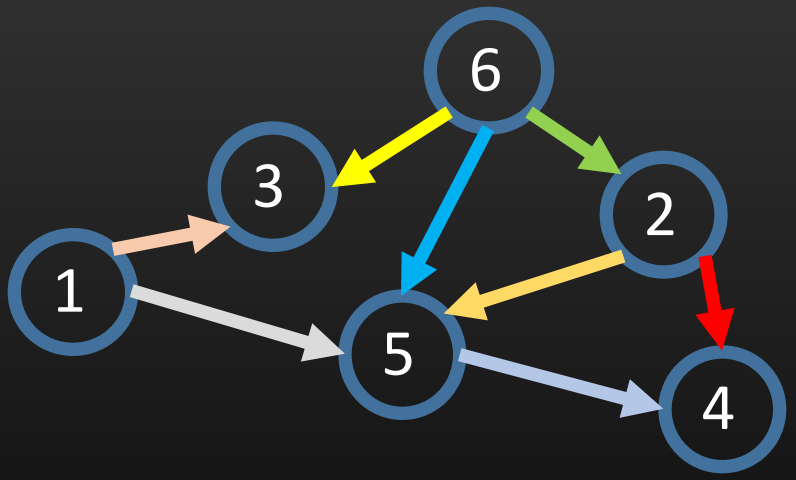


Content

- Graph Adjacency Representation
  - Adjacency Matrix
  - Edge List
  - Adjacency List
- Grid Graph
- Depth-First Search
  - Flood Fill
- Breadth-First Search
  - Unweighted shortest path
  - Multisource BFS
- States

Adjacency matrix

- Use a 2D array to store the edges
- A[i][j] = 0 if there are no edges from vertex i to vertex j
- A[i][j] = 1 if there are edges from vertex i to vertex j

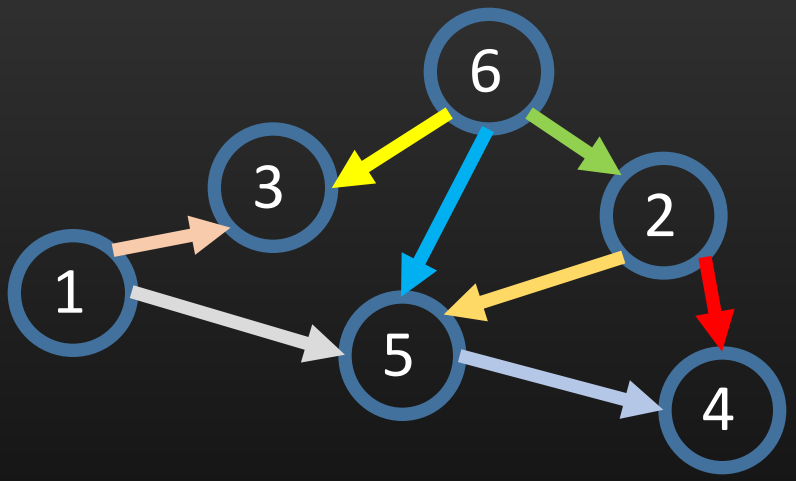


A	1	2	3	4	5	6
1	0	0	1	0	1	0
2	0	0	0	1	1	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	1	0	0
6	0	1	1	0	1	0

- Memory Complexity : $O(|V|^2)$

Edge List

- A list of edges

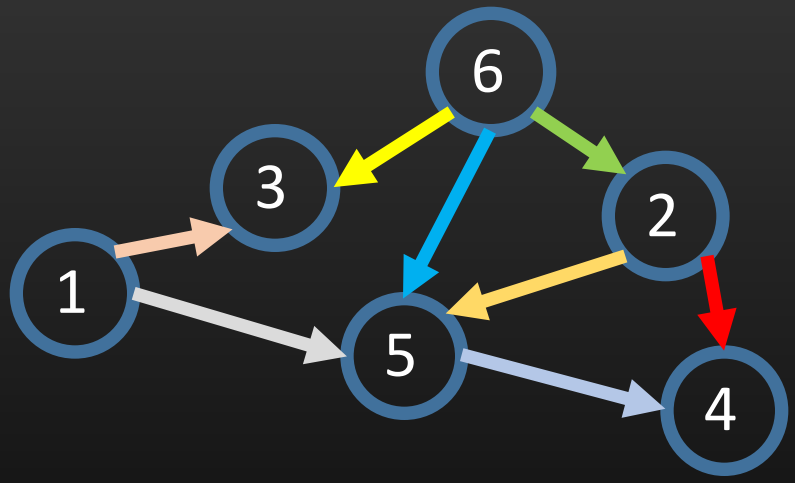


id	x	y
0	1	3
1	6	3
2	1	5
3	2	5
4	2	4
5	6	2
6	5	4
7	6	5

- Memory Complexity : O(|E|)

Edge List

- You may sort the edges in ascending order of x so that all adjacency nodes of a given node can be easily found



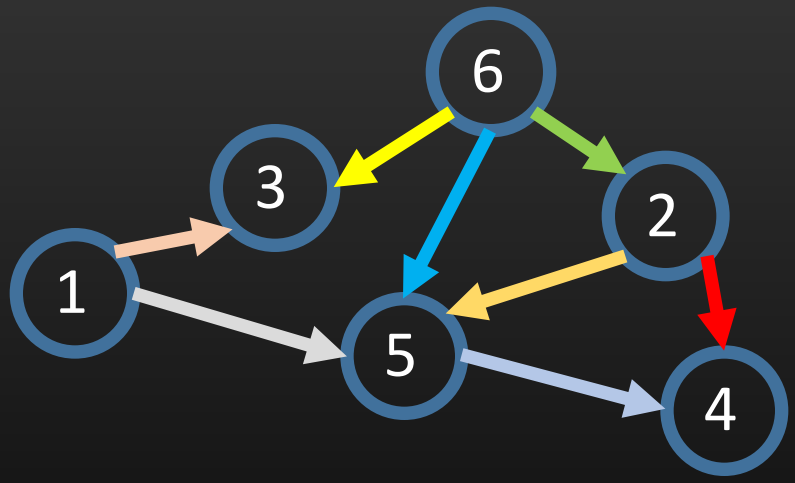
node	first	last
1	0	1
2	2	3
3	4	3
4	4	3
5	4	4
6	5	7

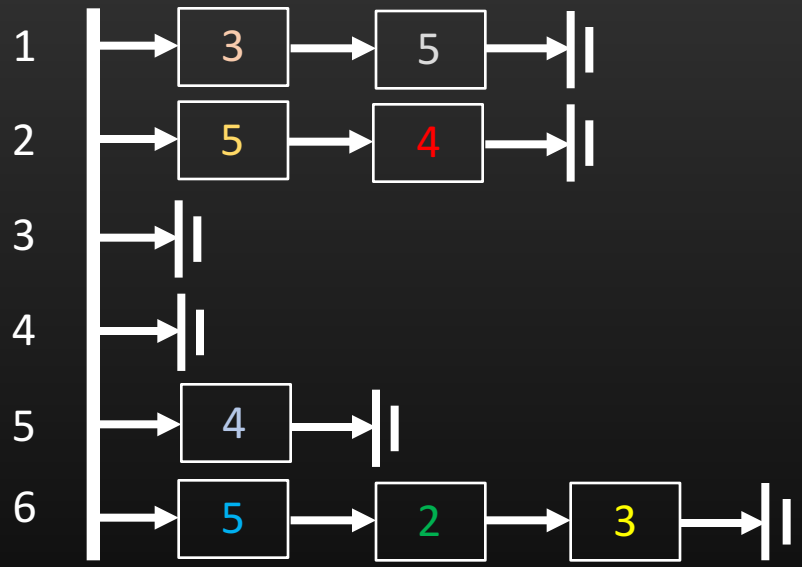
id	x	y
0	1	3
1	1	5
2	2	5
3	2	4
4	5	4
5	6	3
6	6	5
7	6	2

- Time Complexity : O(|E| log |E|) or O(|E|)

Adjacency Lists

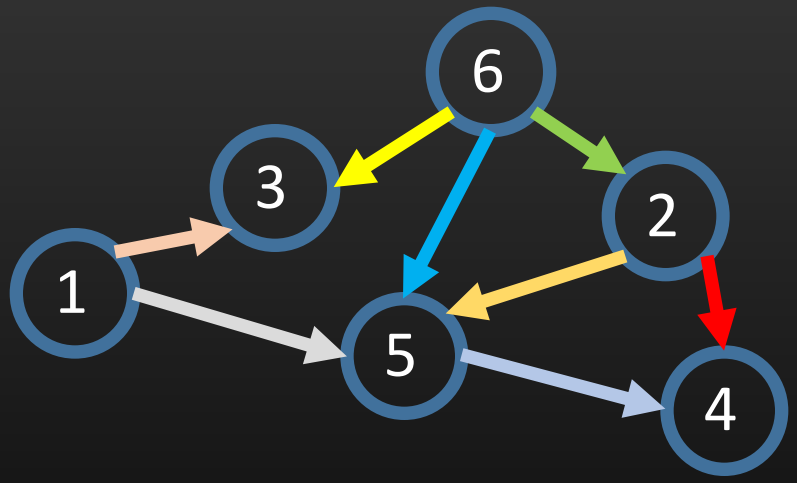
- Use |V| lists to store the edges
- The i-th list stores the vertices vertex i is adjacent to





Adjacency Lists

- Can use a 2D array to imitate the |V| lists

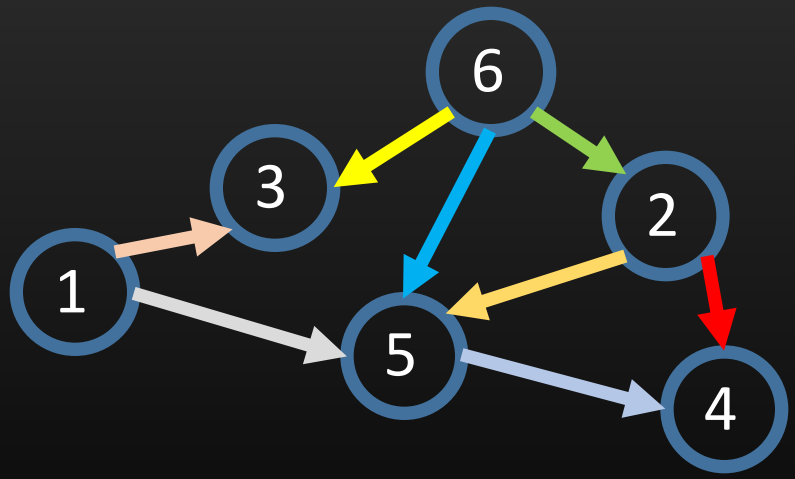


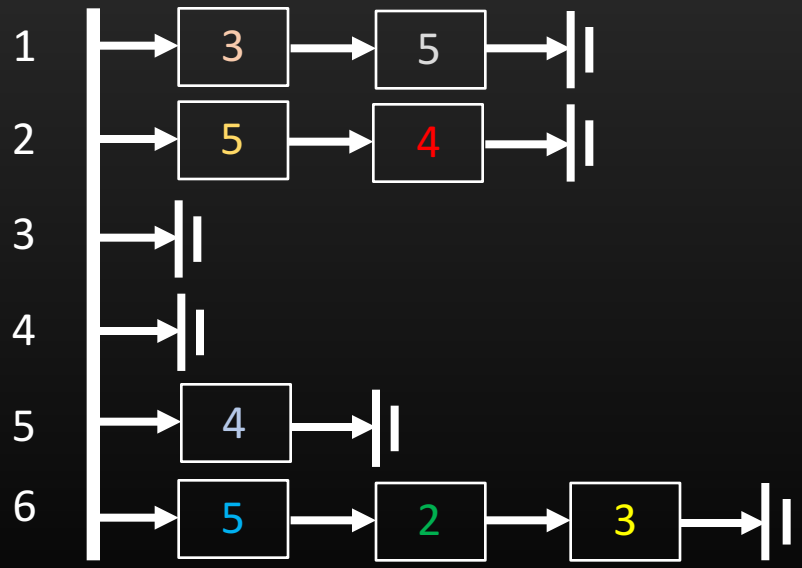
A			
1	3		
2	5	4	
3			
4			
5	4		
6	2	5	3

- Memory Complexity: $O(|V|^2)$

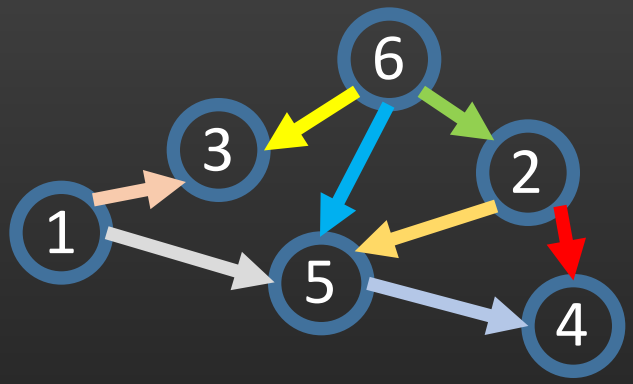
Adjacency Lists

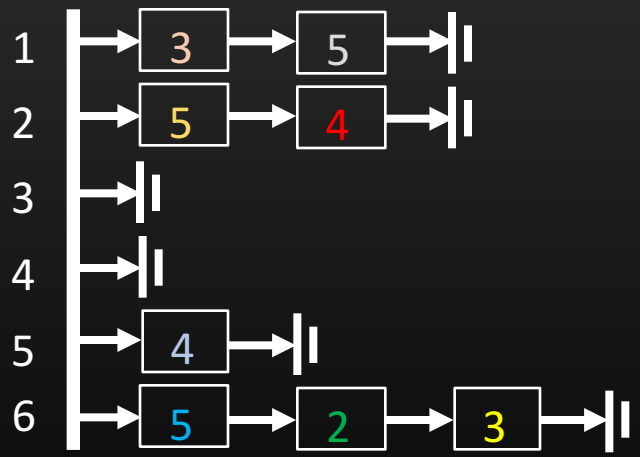
- |E| is usually much less than $|V|^2$ when |V| is large
- Use |V| linked lists to store the adjacent nodes instead
- Memory complexity: O(|E|)





Adjacency Lists - Implementation



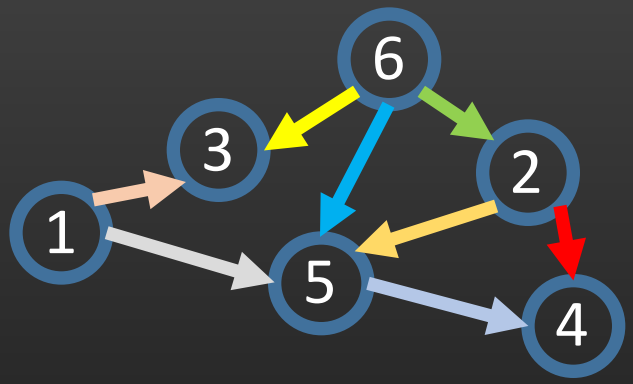


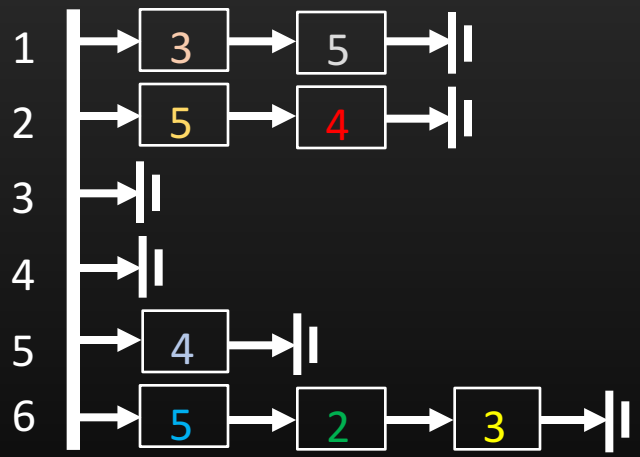
node	first
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1

id	y	next

```
void initialize(){
    for (int i=1;i<=n;i++) first[i]=-1;
}
```

Adjacency Lists - Implementation





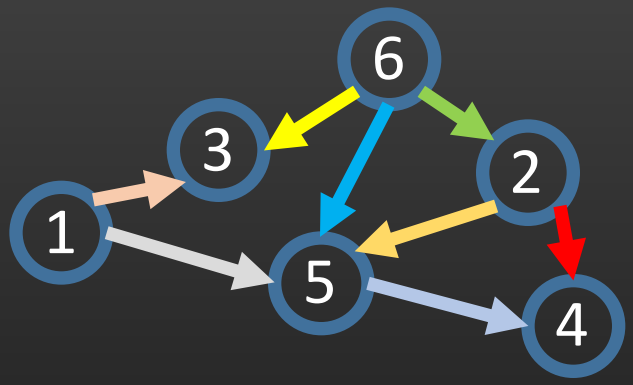
- 1->3

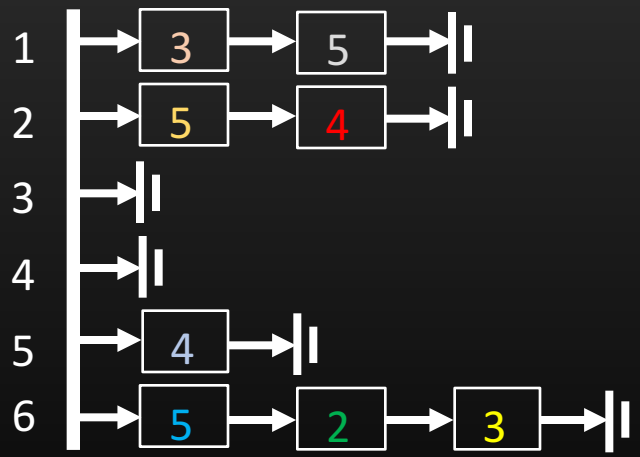
node	first
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1

id	y	next
0	3	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





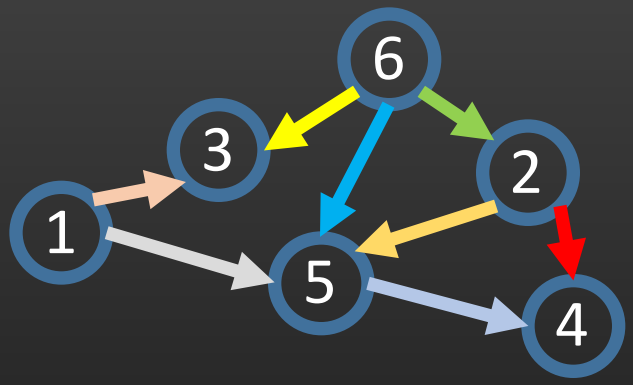
- 1->3

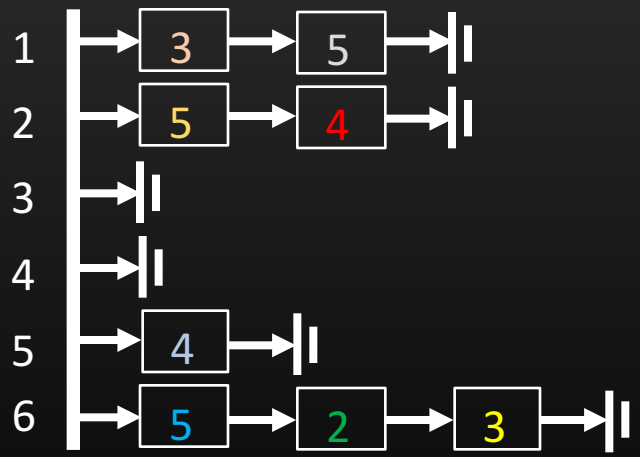
node	first
1	0
2	-1
3	-1
4	-1
5	-1
6	-1

id	y	next
0	3	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





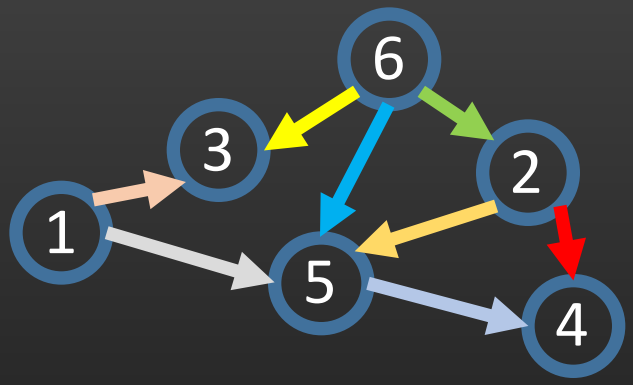
- 6->3

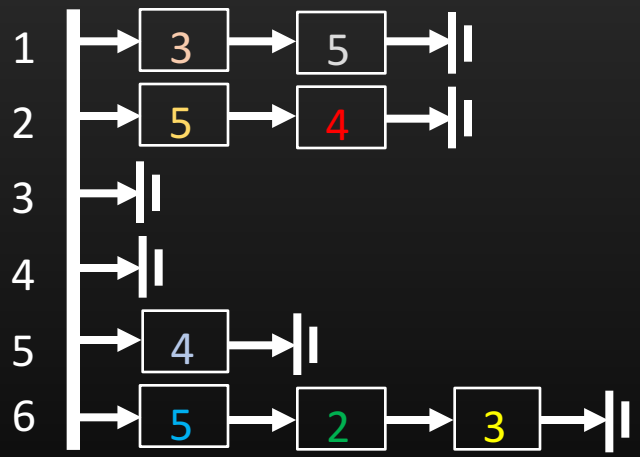
node	first
1	0
2	-1
3	-1
4	-1
5	-1
6	-1

id	y	next
0	3	-1
1	3	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





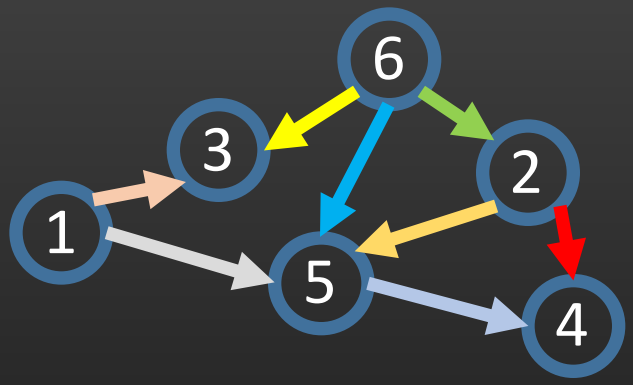
- 6->3

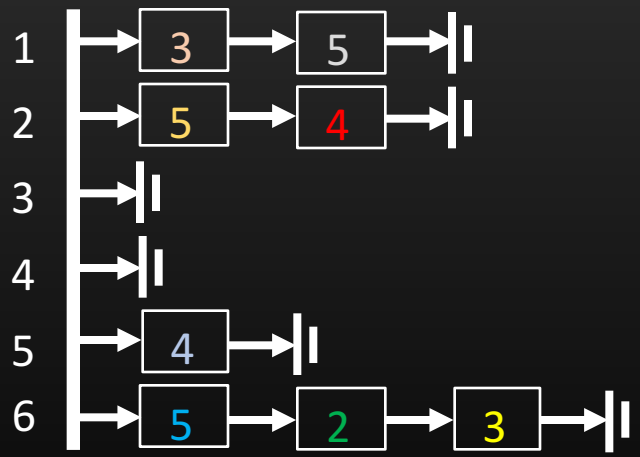
node	first
1	0
2	-1
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





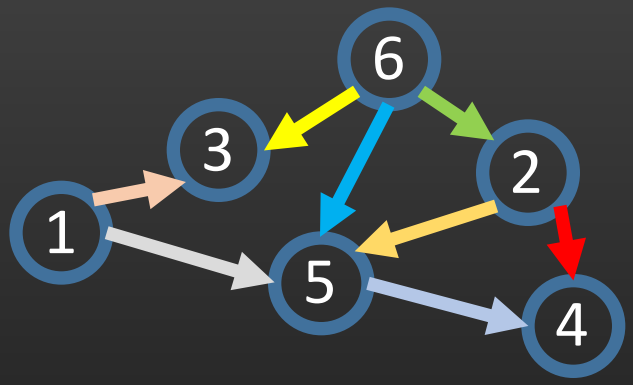
- 1->5

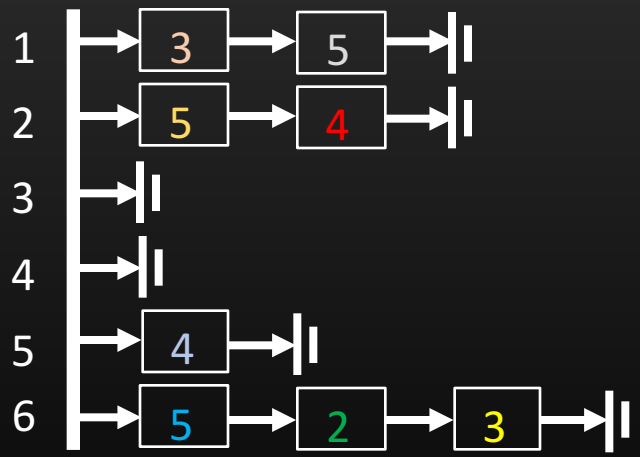
node	first
1	0
2	-1
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





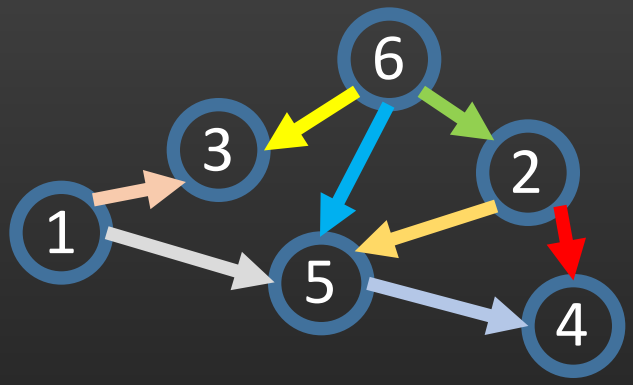
- 1->5

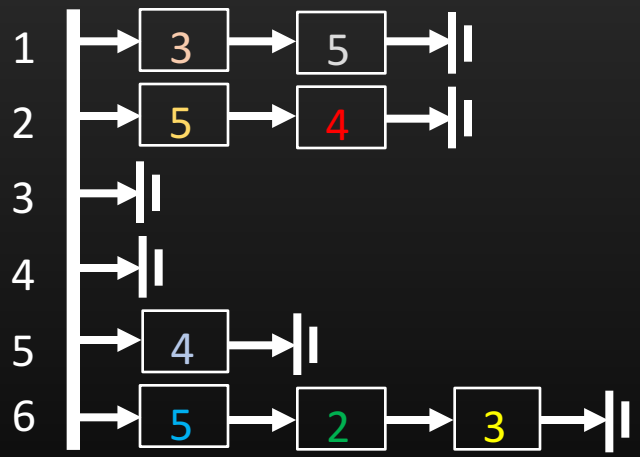
node	first
1	2
2	-1
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





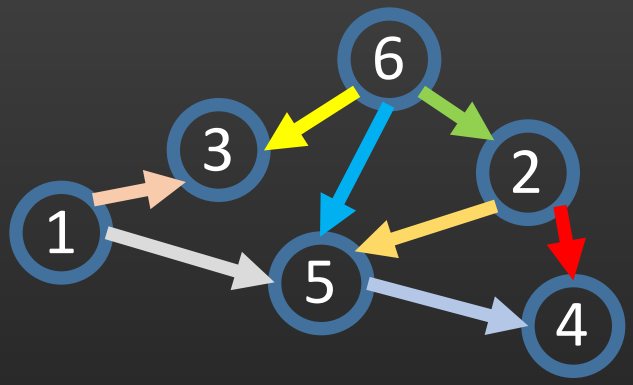
- 2->5

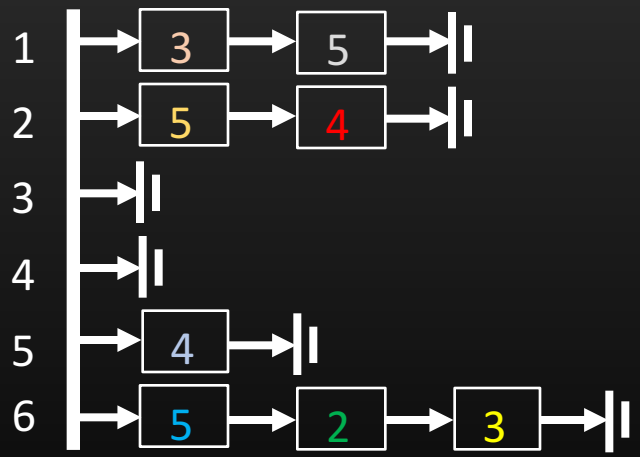
node	first
1	2
2	-1
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





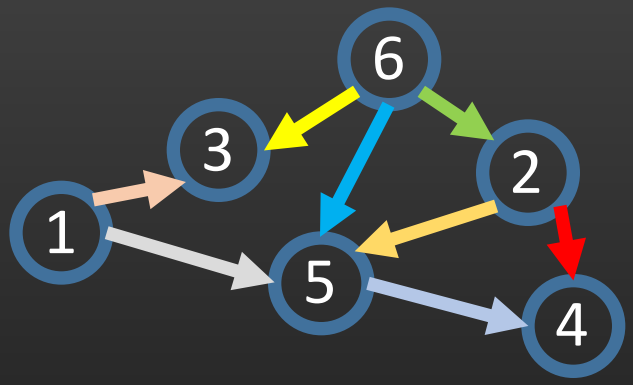
- 2->5

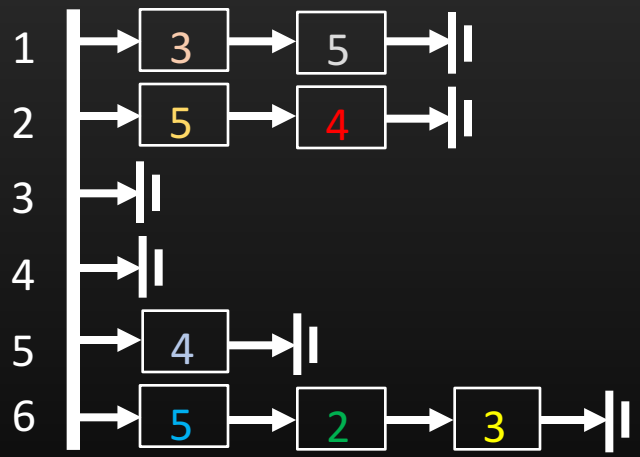
node	first
1	2
2	3
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





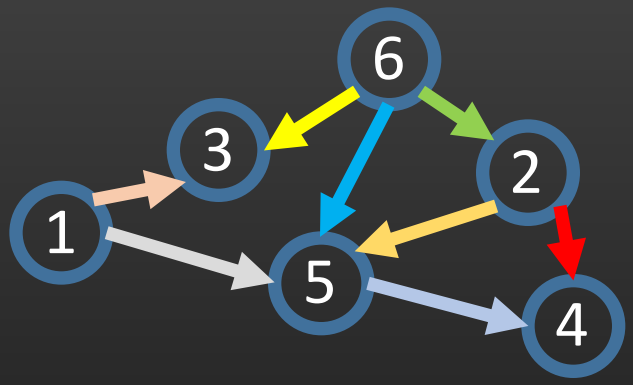
- 2->4

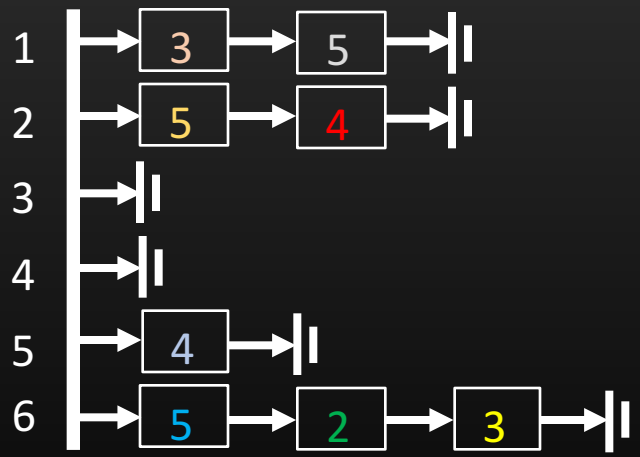
node	first
1	2
2	3
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





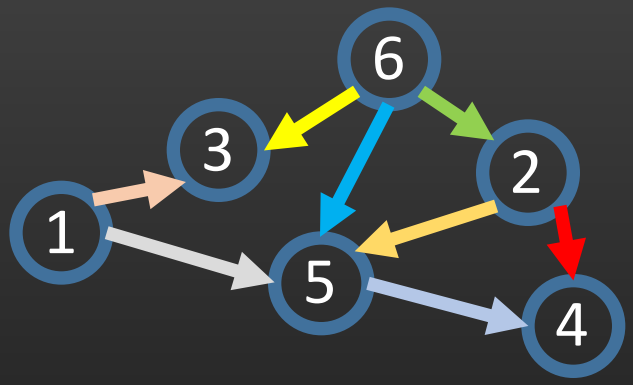
- 2->4

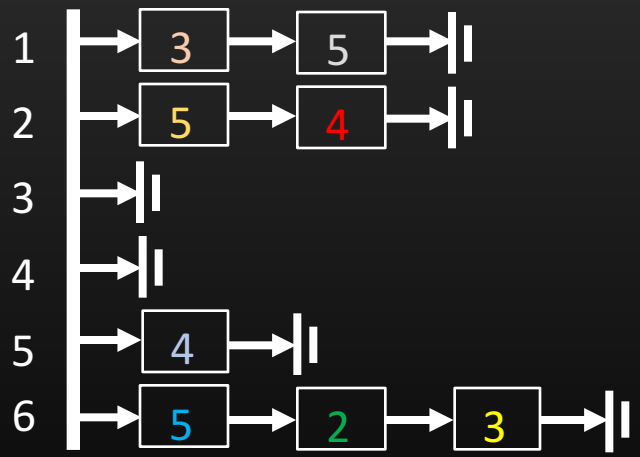
node	first
1	2
2	4
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





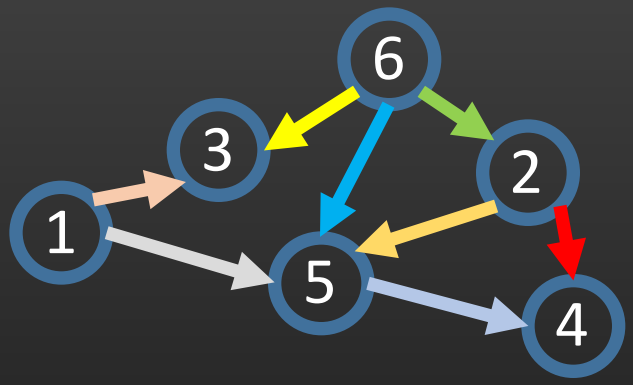
- 6->2

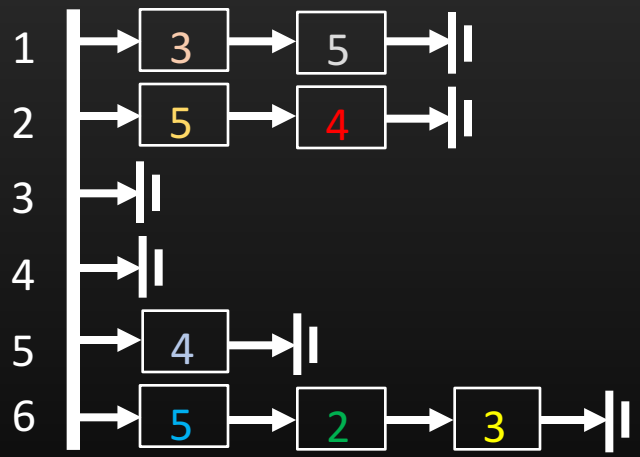
node	first
1	2
2	4
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





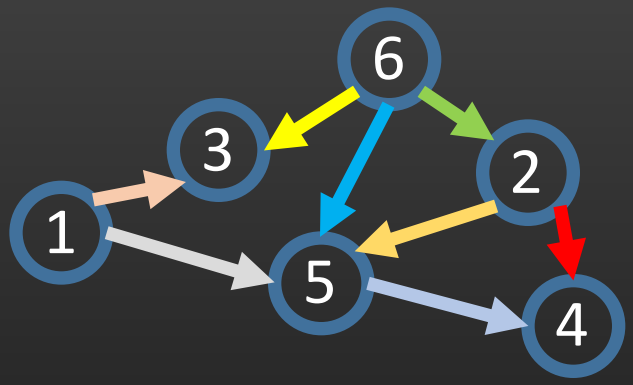
- 6->2

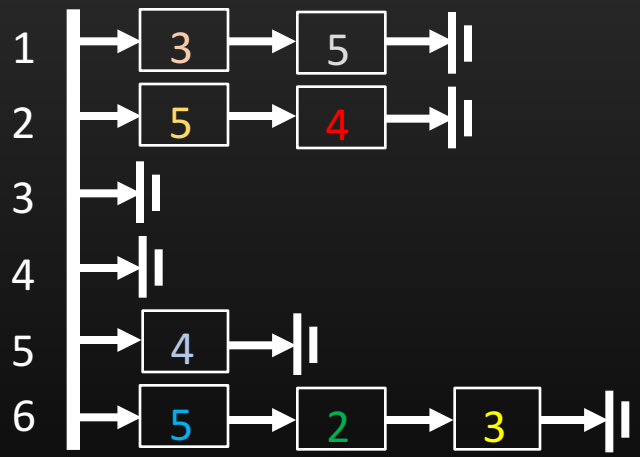
node	first
1	2
2	4
3	-1
4	-1
5	-1
6	5

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





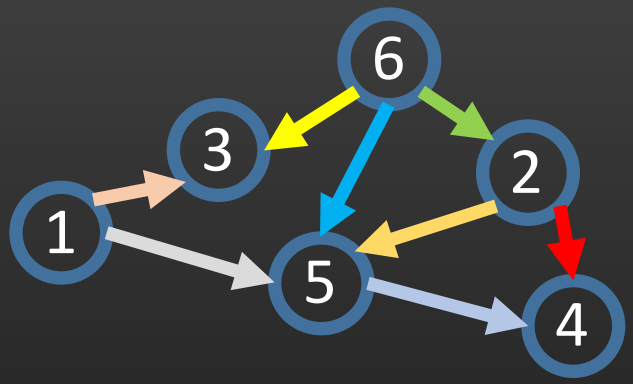
- 5->4

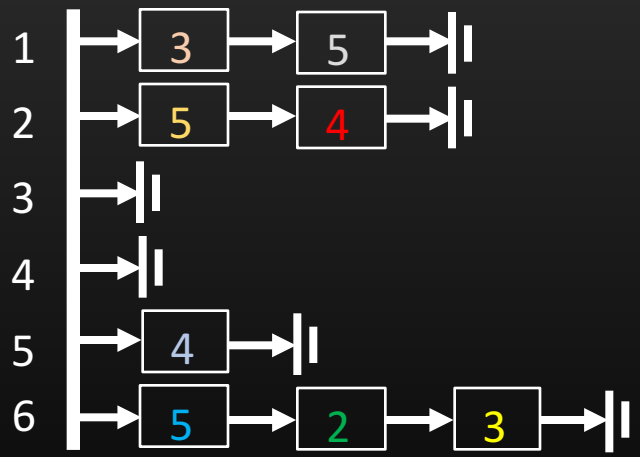
node	first
1	2
2	4
3	-1
4	-1
5	6
6	5

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1
6	4	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





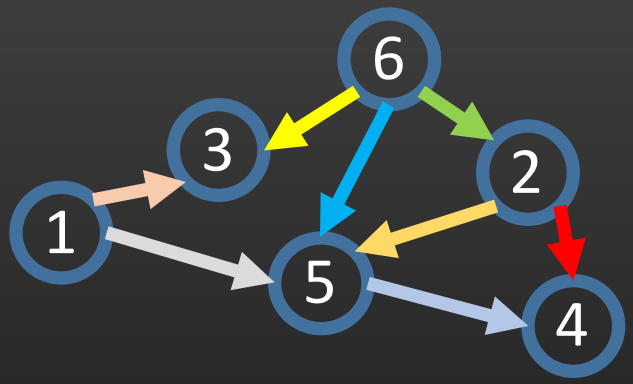
- 6->5

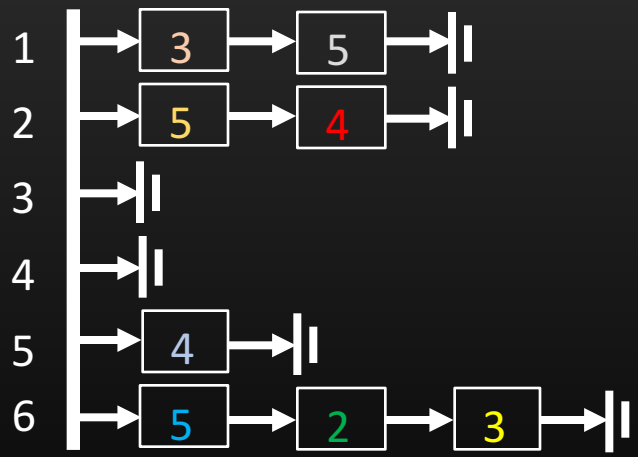
node	first
1	2
2	4
3	-1
4	-1
5	6
6	7

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1
6	4	-1
7	5	5

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

Adjacency Lists - Implementation





- Query 6's neighbors

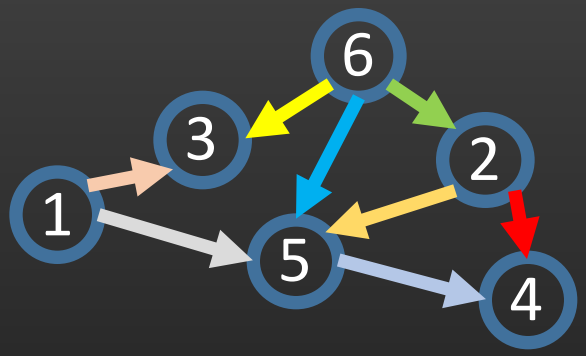
node	first
1	2
2	4
3	-1
4	-1
5	6
6	7

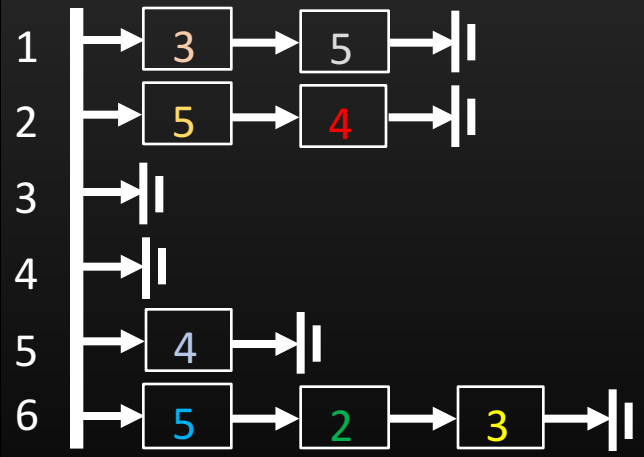
id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1
6	4	-1
7	5	5

```
void query(int x){
    //Querying all adjacent nodes of a given node x
    for (int i=first[x];i!=-1;i=next[i])
        printf("%d\n",v[i]);
}
```

Adjacency Lists - Implementation

```
int v[10000],w[10000],next[10000];
int first[1000];
int n,m=0;

void initialize(){
    for (int i=1;i<=n;i++) first[i]=-1;
}

void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}

void query(int x){
    //Querying all adjacent nodes of a given node x
    for (int i=first[x];i!=-1;i=next[i])
        printf("%d\n",v[i]);
}
```

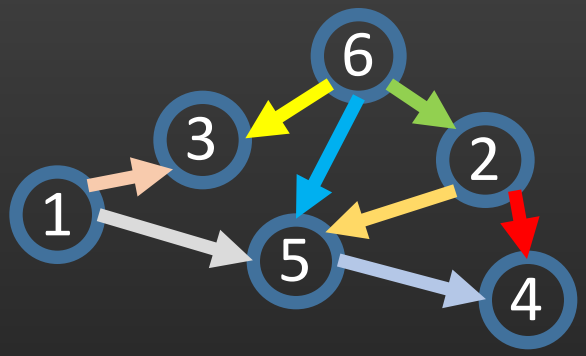


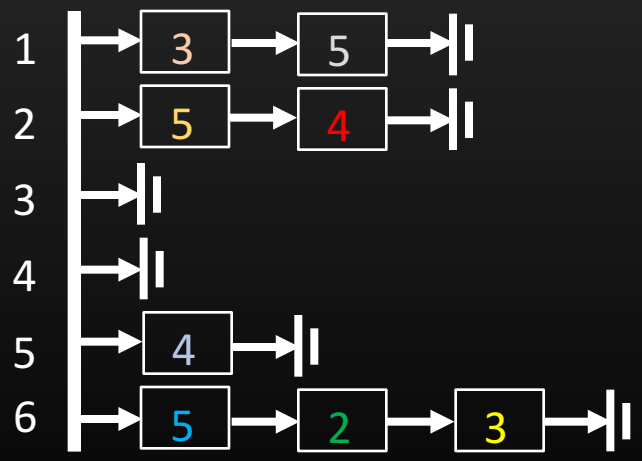


Adjacency Lists - Implementation

- We can use vector in C++ to implement the lists

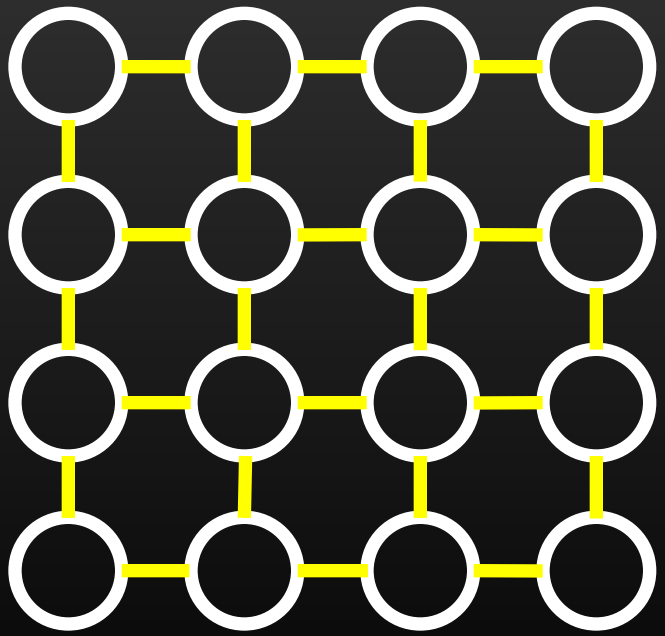
```cpp
#include<vector>
using namespace std;

int n,m;
vector<int> v[10000],w[10000];
void new_edge(int from,int to,int weight){
    v[from].push_back(to);
    w[from].push_back(weight);
}

void query(int x){
    //Querying all adjacent nodes of a given node (x)
    for (int i=0;i<v[x].size();i++)
        printf("%d\n",v[x][i]);
}
```





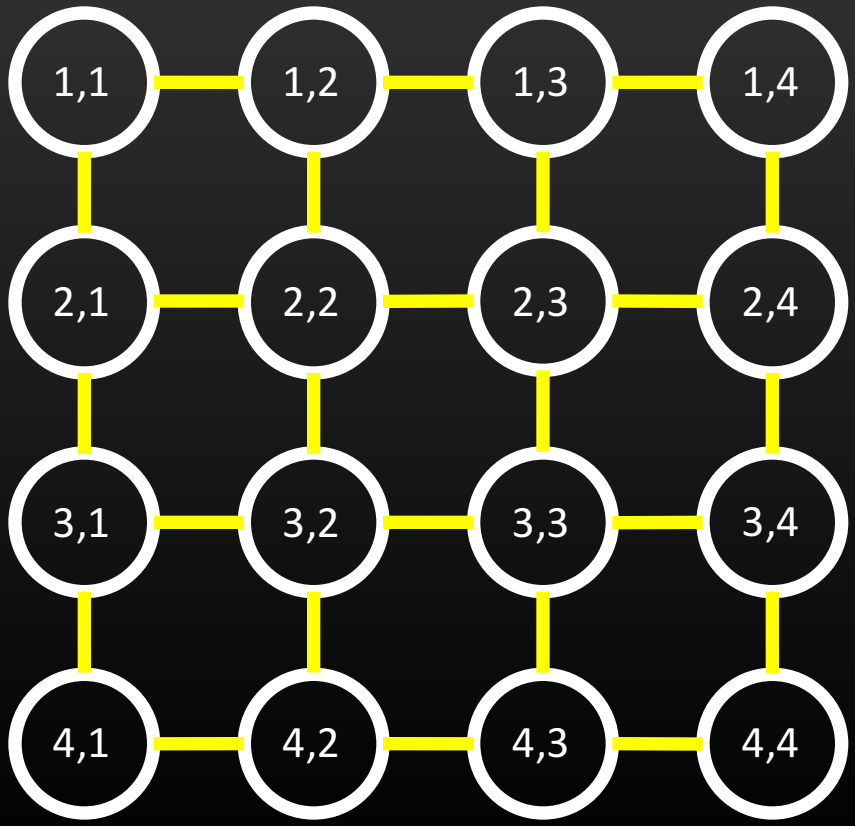
Grid Graph

- A graph that the vertices form a regular tiling
- E.g. rectangles, cubes or triangles



Grid Graph - Vertex

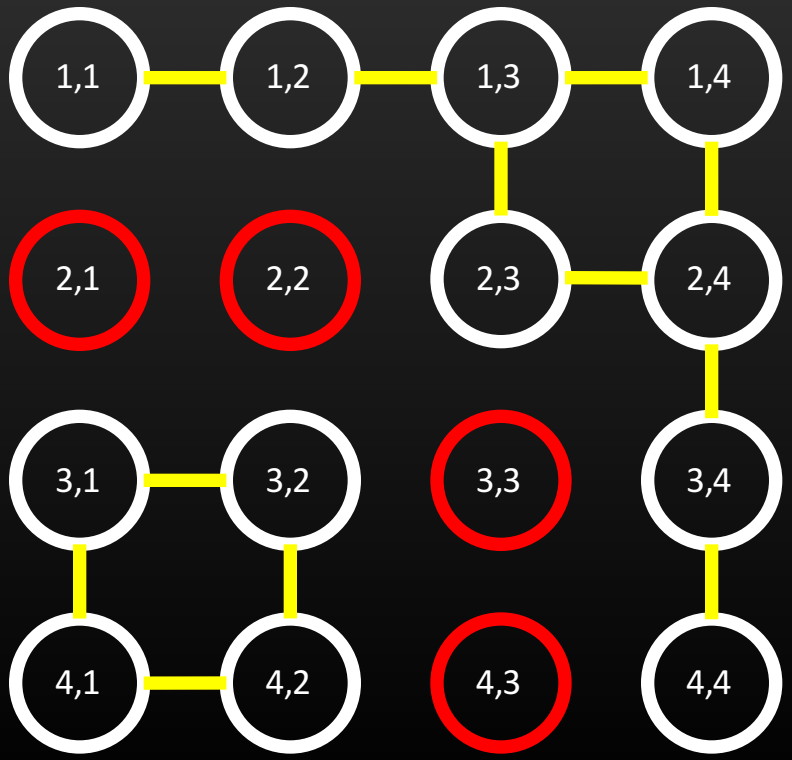
- No need to give new indices to the vertices
- Use their coordinates instead



Grid Graph - Vertex

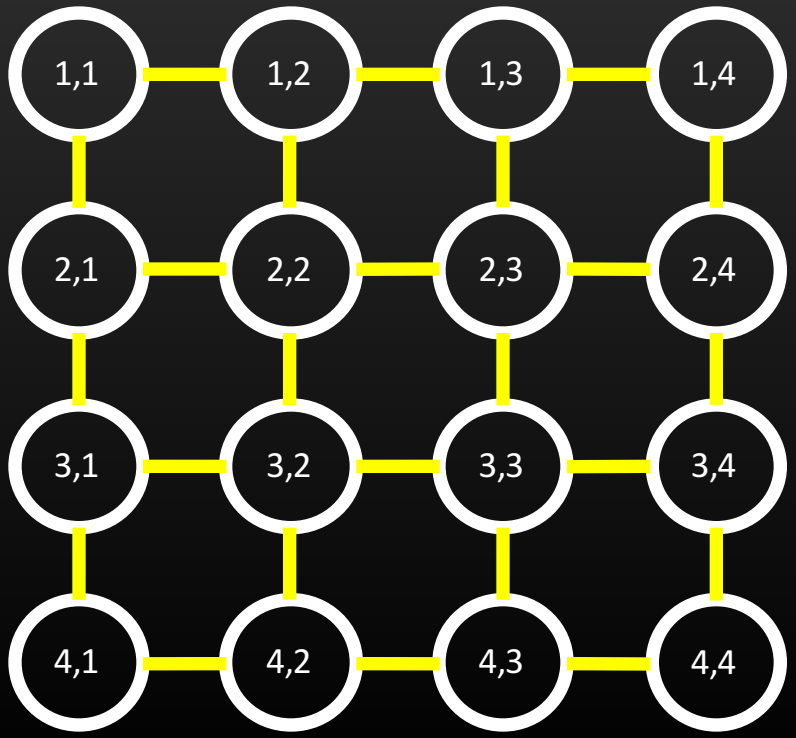
- Some vertices may be invalid (e.g. a wall in a maze)
- Skip the process when visiting an invalid vertex

```
int valid(int x, int y){
    if (x <= 0 || x > n) return 0;
    if (y <= 0 || y > m) return 0;
    if (isWall[x][y]) return 0;
    ...
    return 1;
}
```



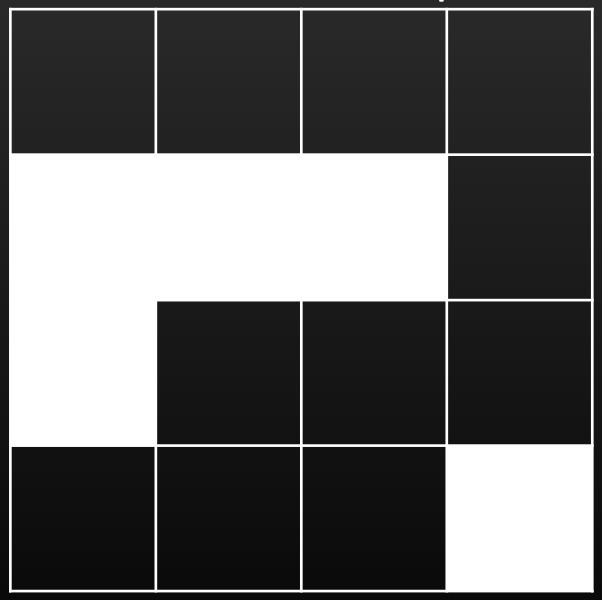
Grid Graph - Edge

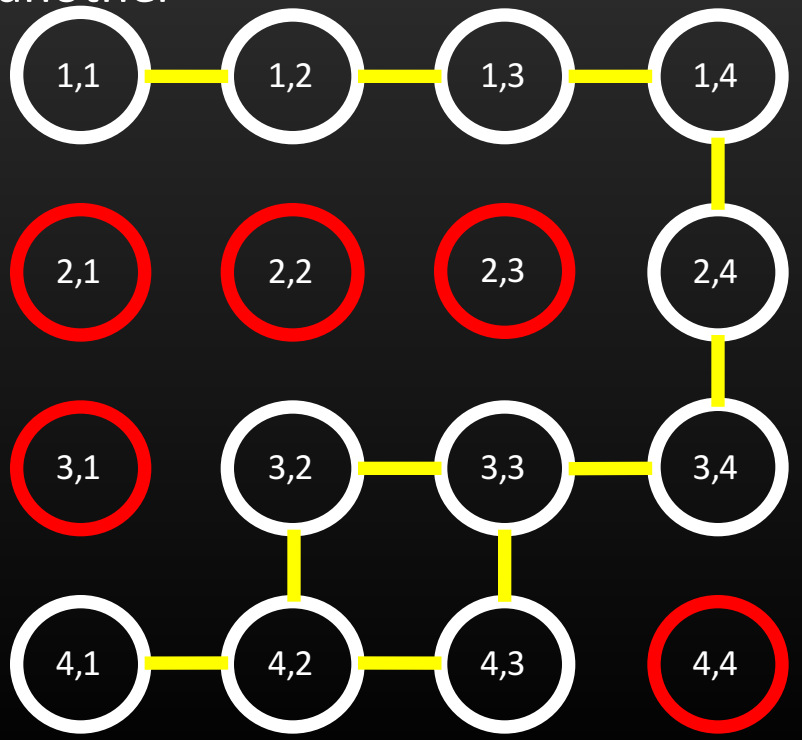
- The number of edges for each vertex is small
- No need to use the previous representations to store the edges
- The coordinates of the adjacent vertices can be calculated



Grid Graph – Example 1

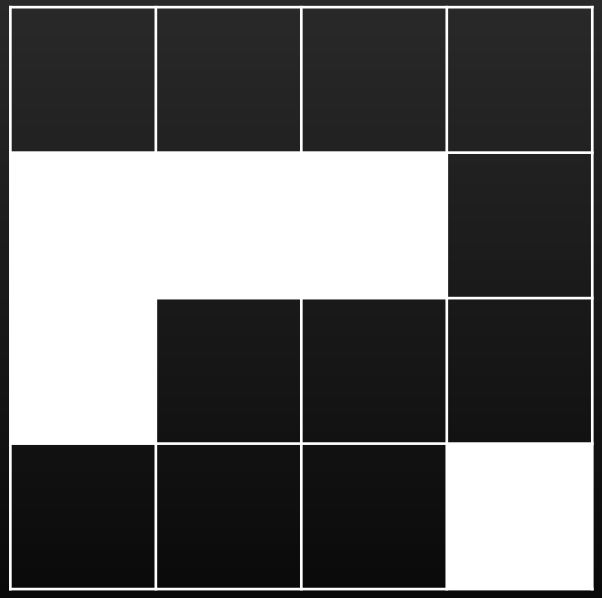
- A rectangular maze
- You are only allowed to move one cell horizontally or vertically in one move
- Find the shortest path from one to another

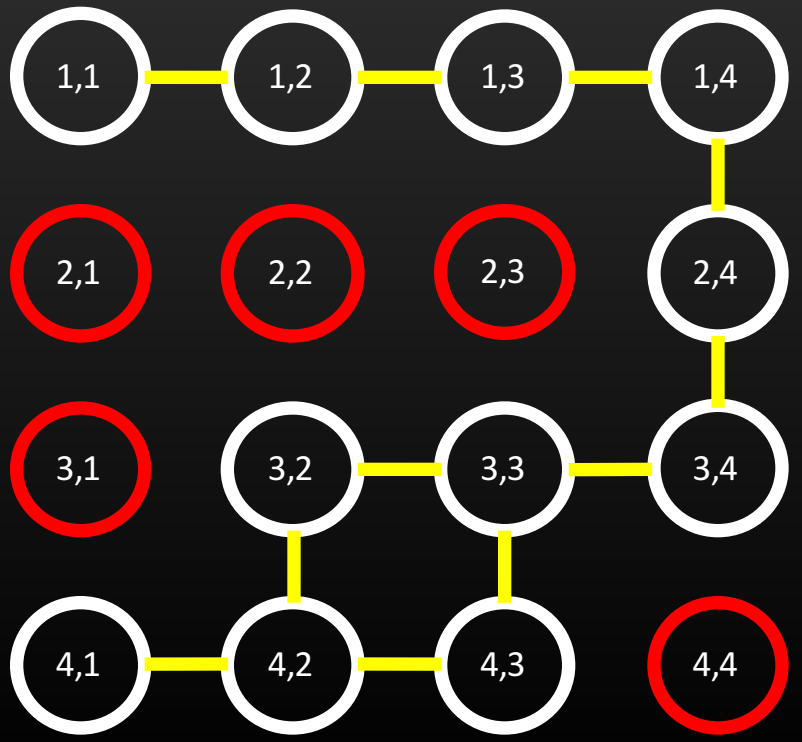




Grid Graph – Example 1

- The possible adjacent vertices of vertex (x, y) are (x – 1, y), (x + 1, y), (x, y – 1) and (x, y + 1)



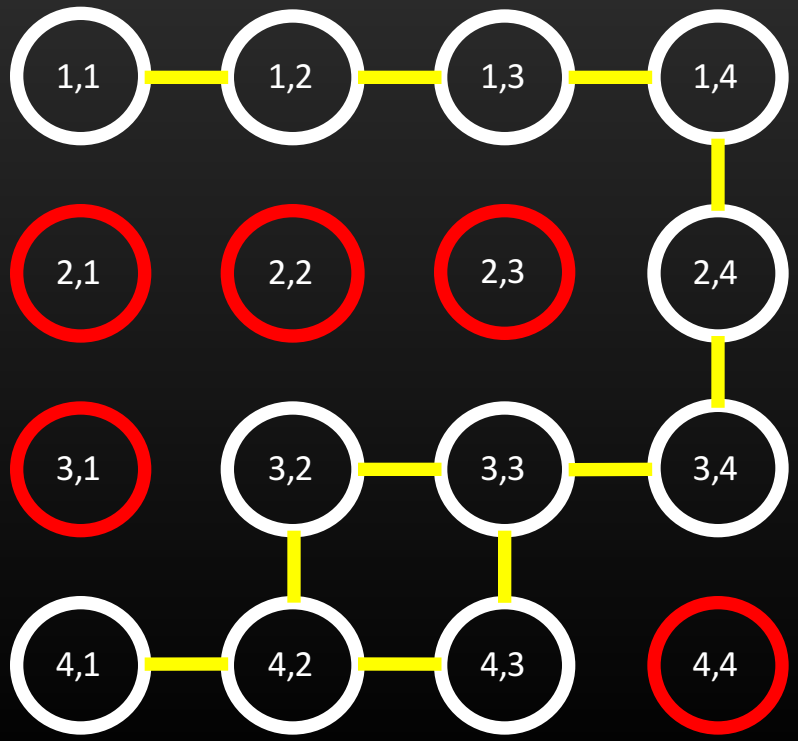


Grid Graph – Example 1

- Hardcode the edges

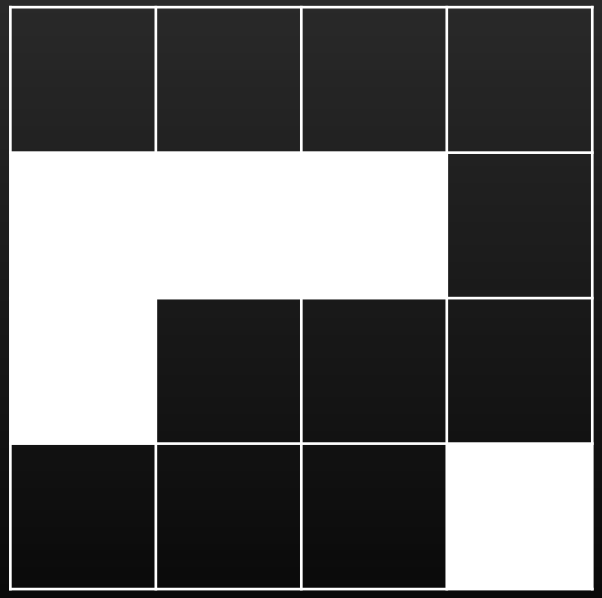
```
int dx[4]={-1,1,0,0};
int dy[4]={0,0,-1,1};

...
// find nodes adjacent to (x, y)
for (int i = 0; i < 4; i++){
    newX = x + dx[i];
    newY = y + dy[i];
    if (!valid(newX, newY)) continue;
    ...
}
...
```

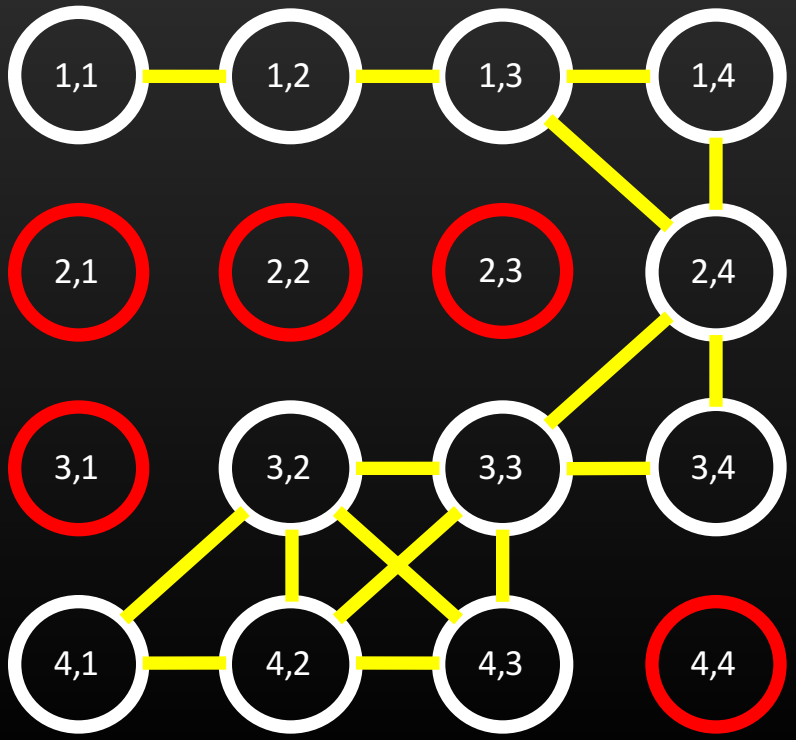
- Use Breath-first Search to find the shortest path



Grid Graph – Example 2

- You are allowed to move one cell horizontally, vertically or diagonally in one move

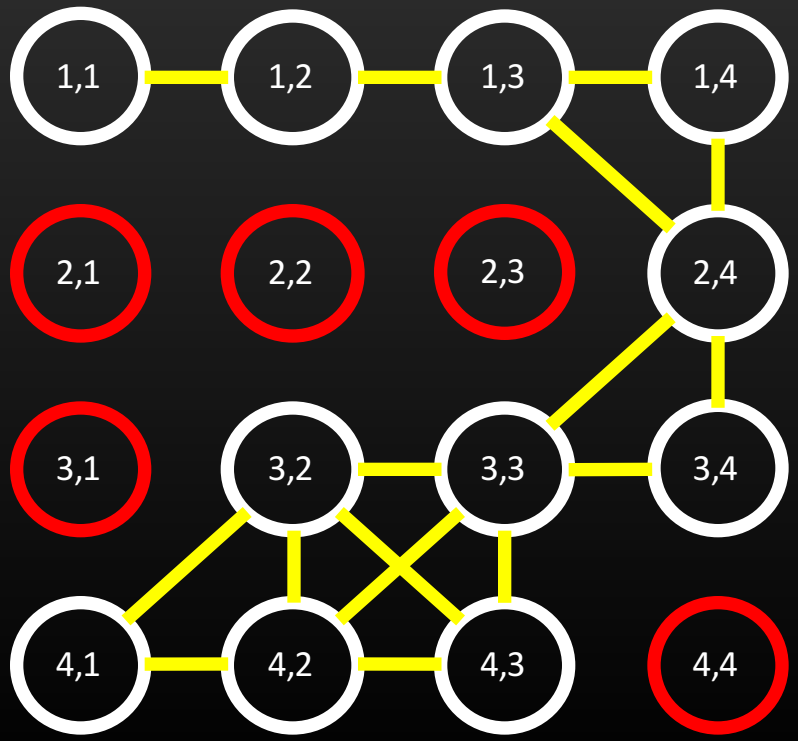




Grid Graph – Example 2

- Hardcode

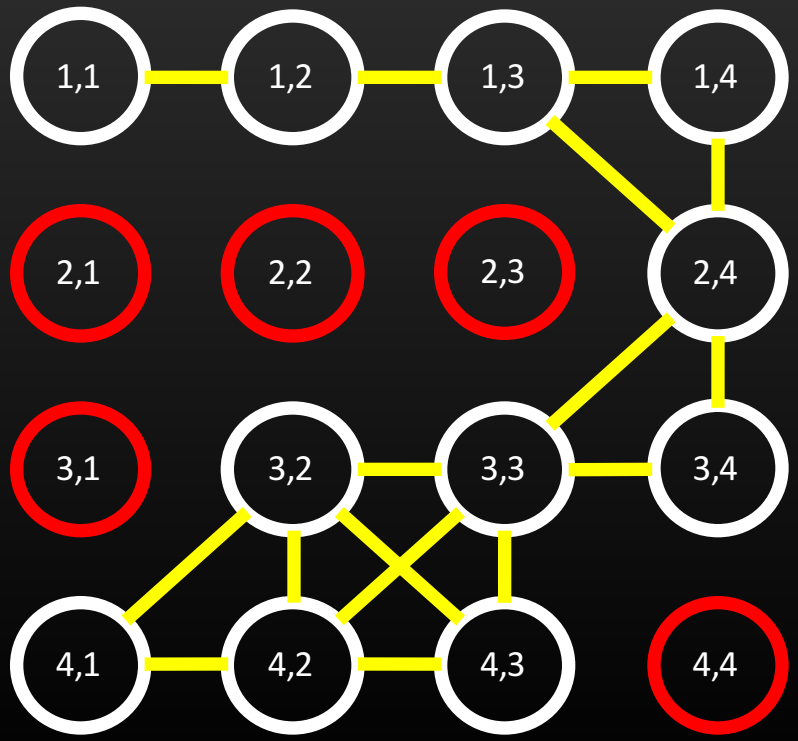
```
int dx[8]={-1,-1,-1,0,0,1,1,1};
int dy[8]={-1,0,1,-1,1,-1,0,1};

...
// find nodes adjacent to (x, y)
for (int i = 0; i < 8; i++){
    newX = x + dx[i];
    newY = y + dy[i];
    if (!valid(newX, newY)) continue;
    ...
}
...
```



Grid Graph – Example 2

- Use 2 for loops

```
// find nodes adjacent to (x, y)
for (int i = -1; i <= 1; i++)
    for (int j = -1; j <= 1; j++){
    newX = x + i;
    newY = y + j;
    if (newX == x && newY == y) continue;
    if (!valid(newX, newY)) continue;
    ...
}
```



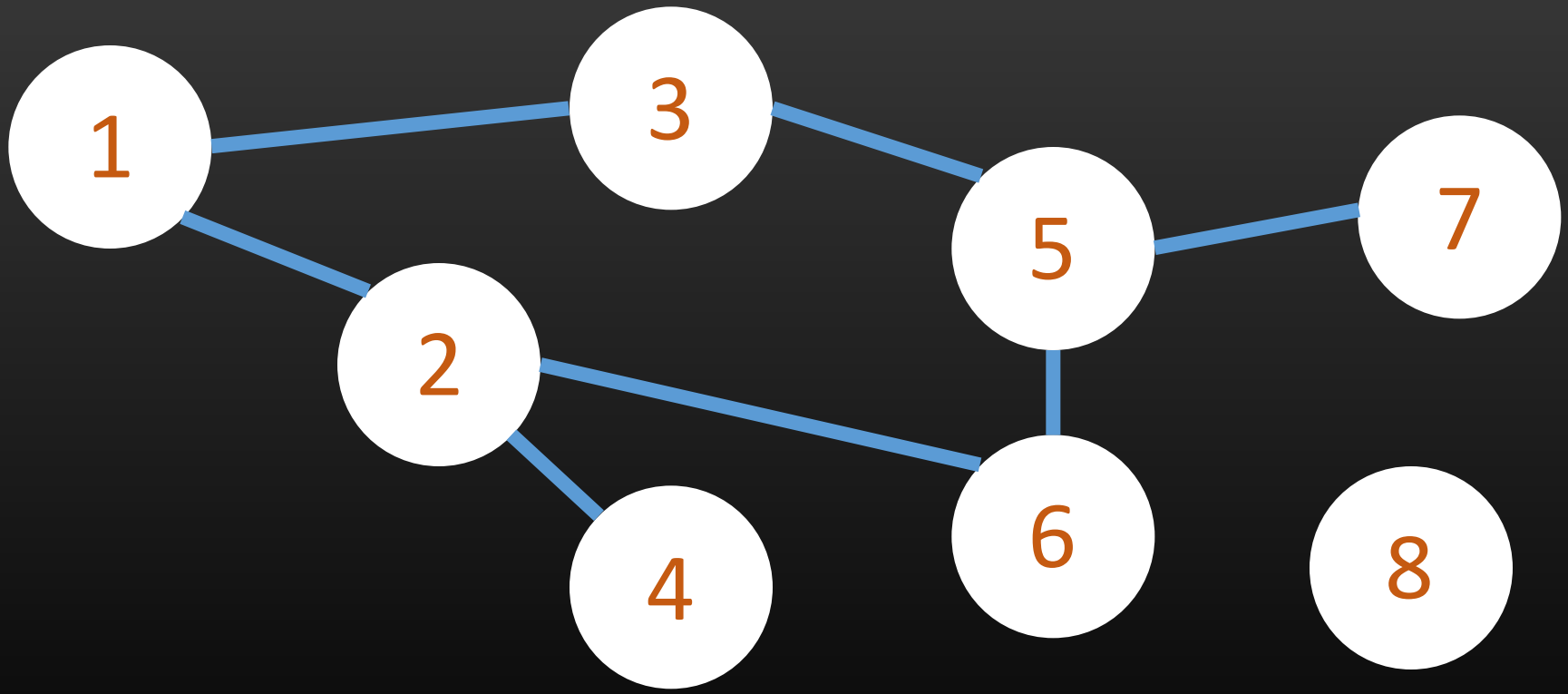
Graph Traversal

- The process of visiting (checking and/or updating) each vertex in a graph.
- Classified by the order in which the vertices are visited
  - Depth-first Search
  - Breadth-first Search

Depth-first Search

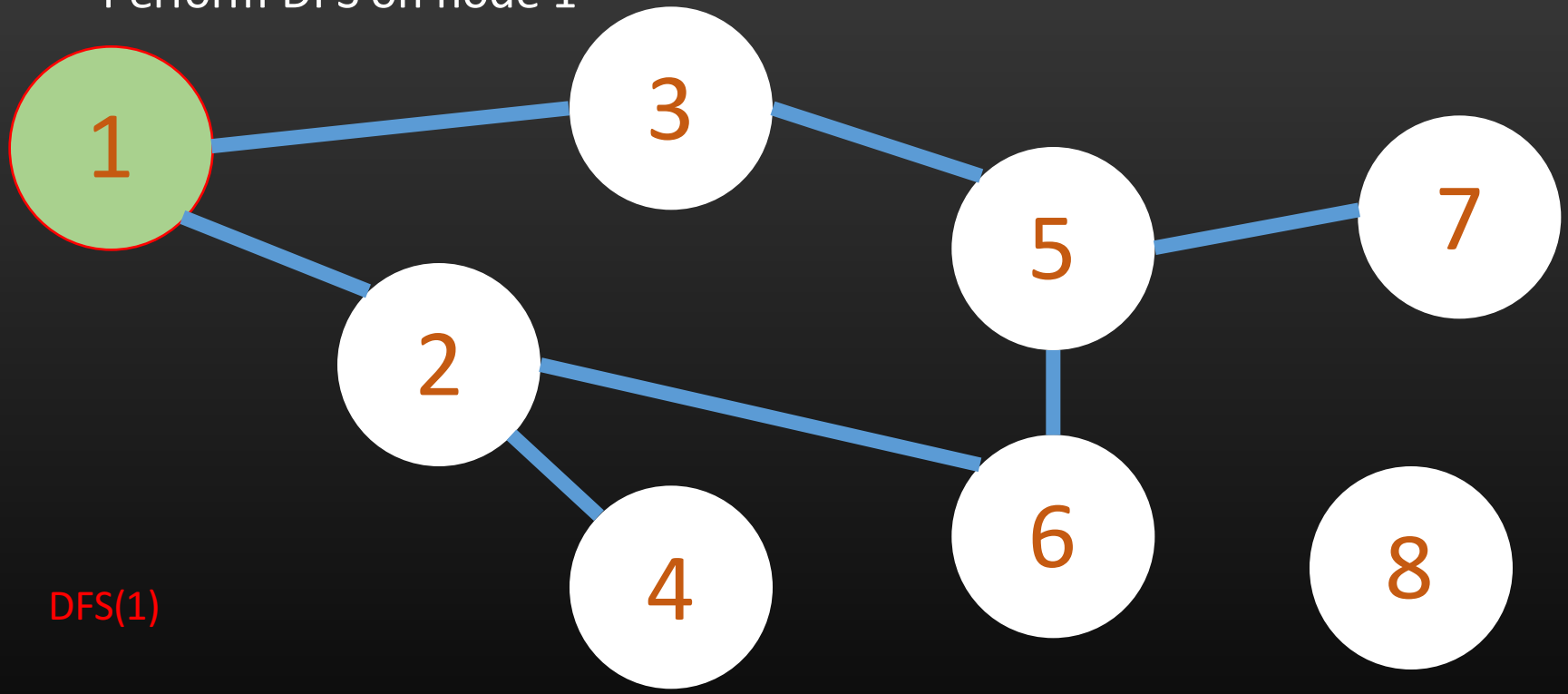
- Starts at a node
- Explores as far as possible along each branch before backtracking
- Recursion is often used

Depth-first Search - example



Depth-first Search - example

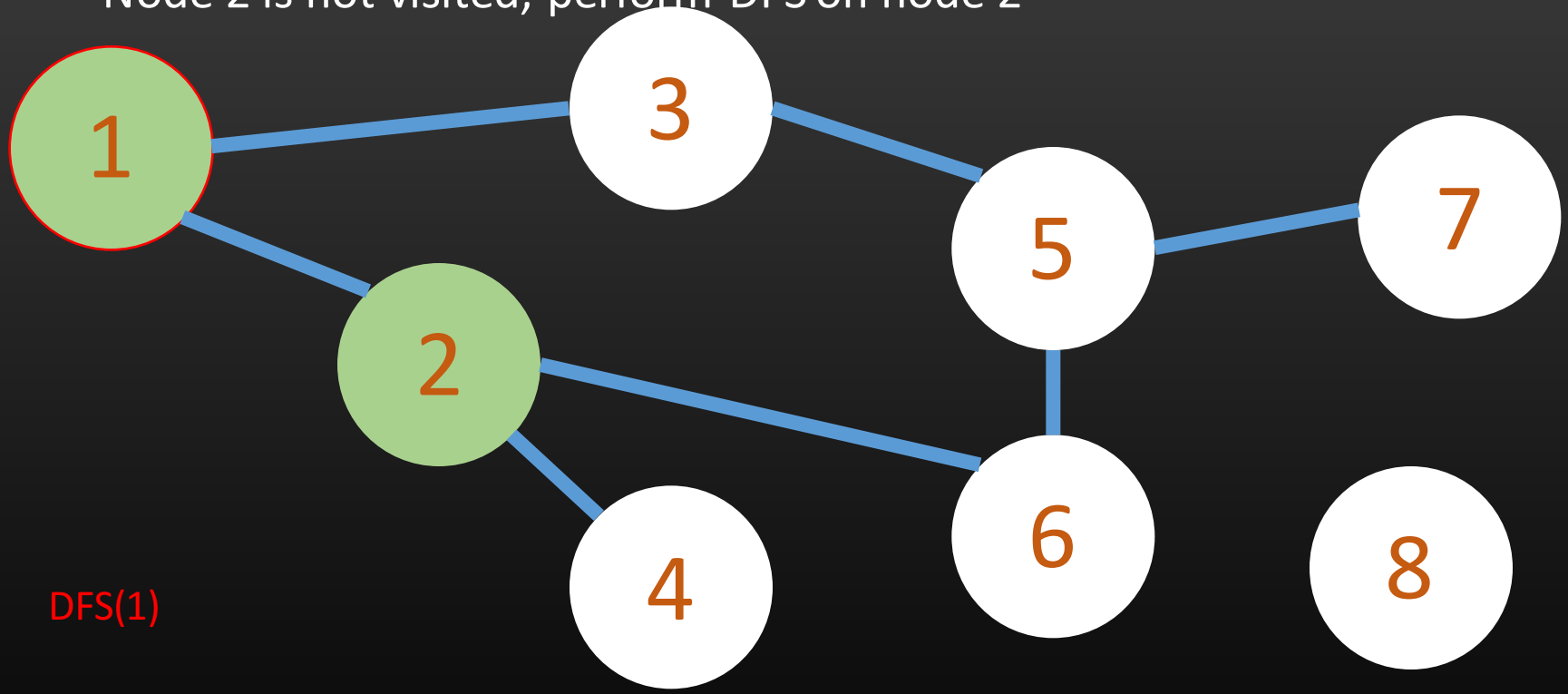
- Perform DFS on node 1



DFS(1)

Depth-first Search - example

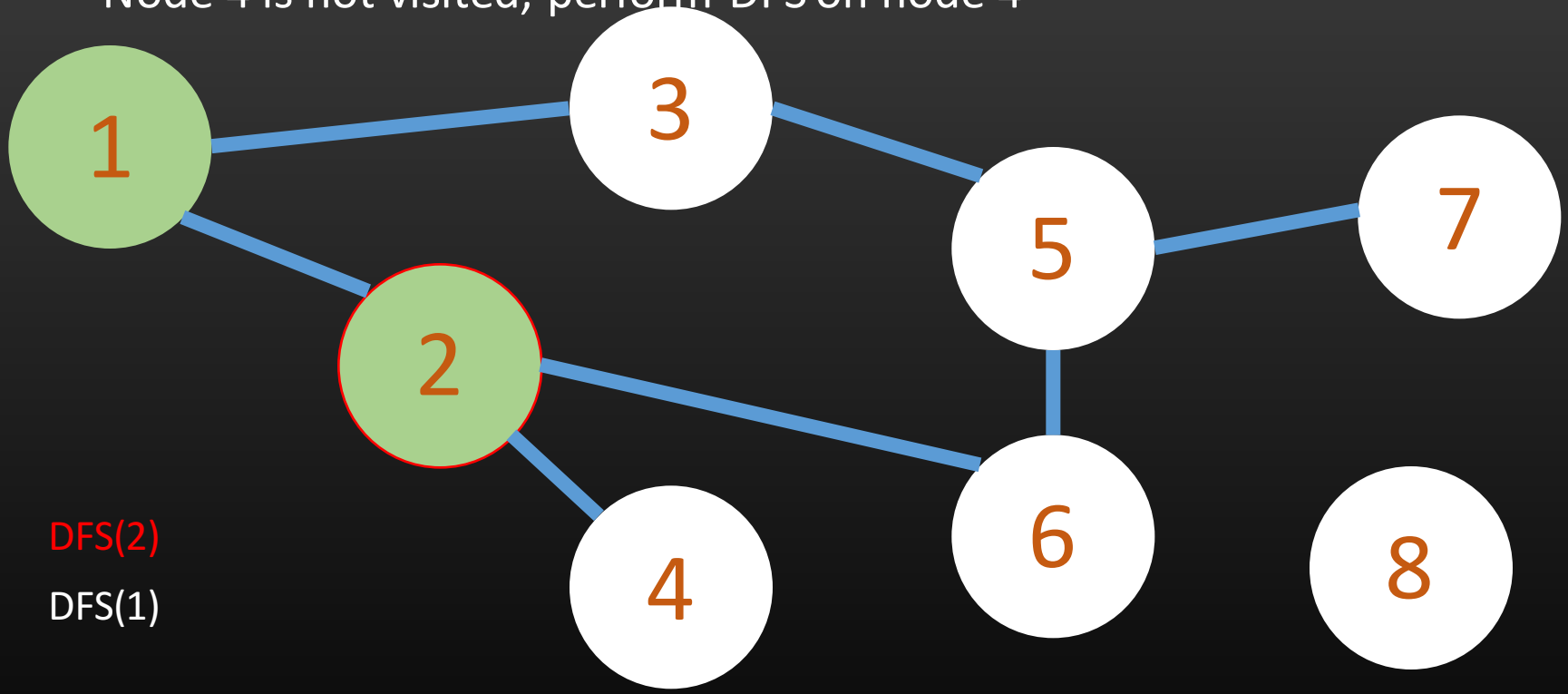
- Node 2 is not visited, perform DFS on node 2



DFS(1)

Depth-first Search - example

- Node 4 is not visited, perform DFS on node 4

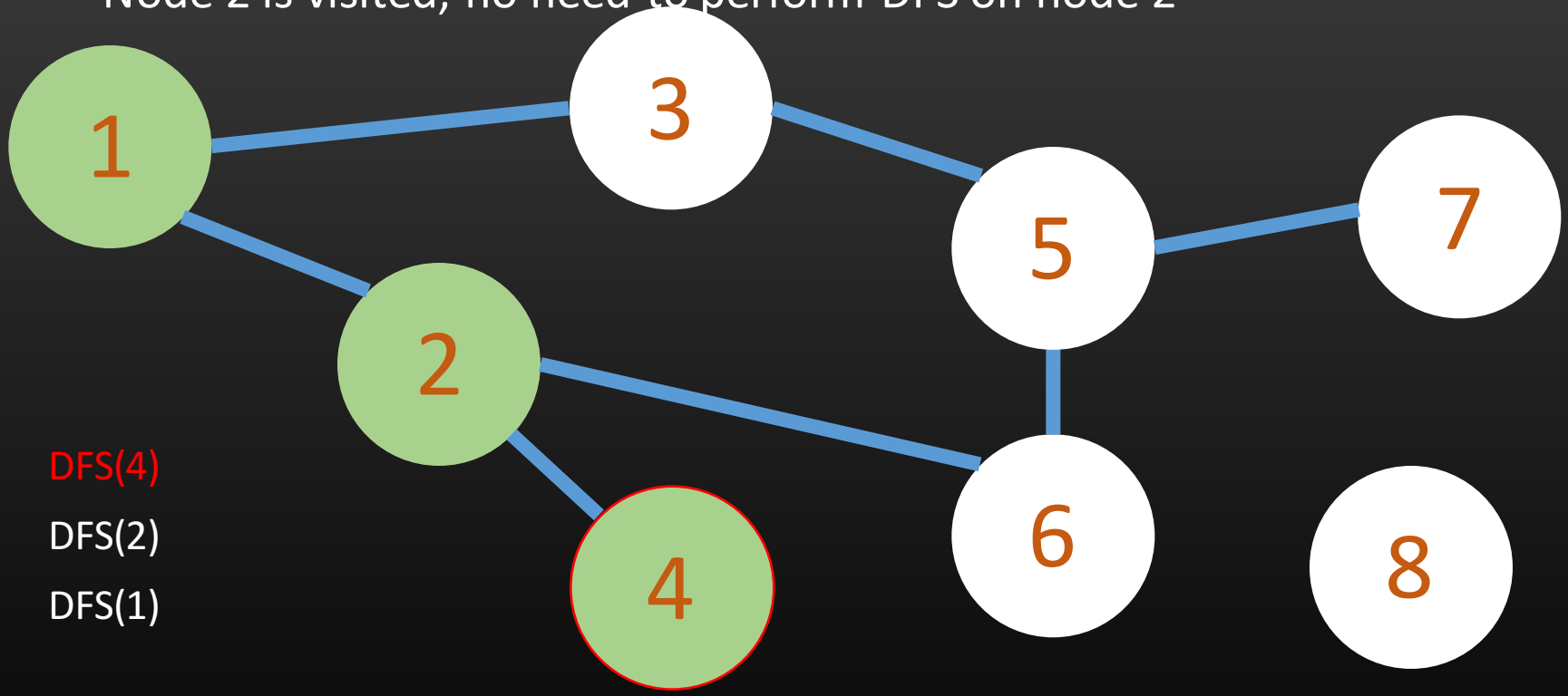


DFS(2)

DFS(1)

Depth-first Search - example

- Node 2 is visited, no need to perform DFS on node 2



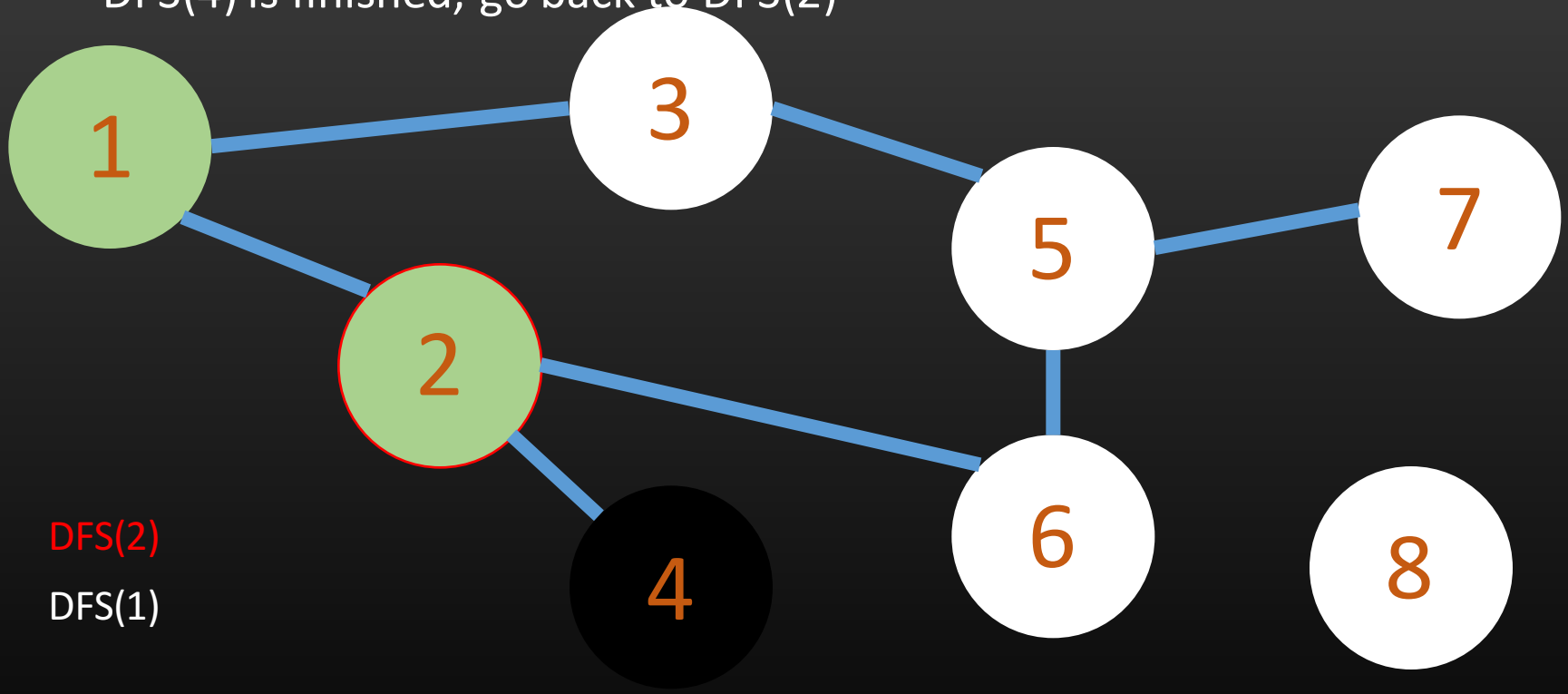
DFS(4)

DFS(2)

DFS(1)

Depth-first Search - example

- DFS(4) is finished, go back to DFS(2)

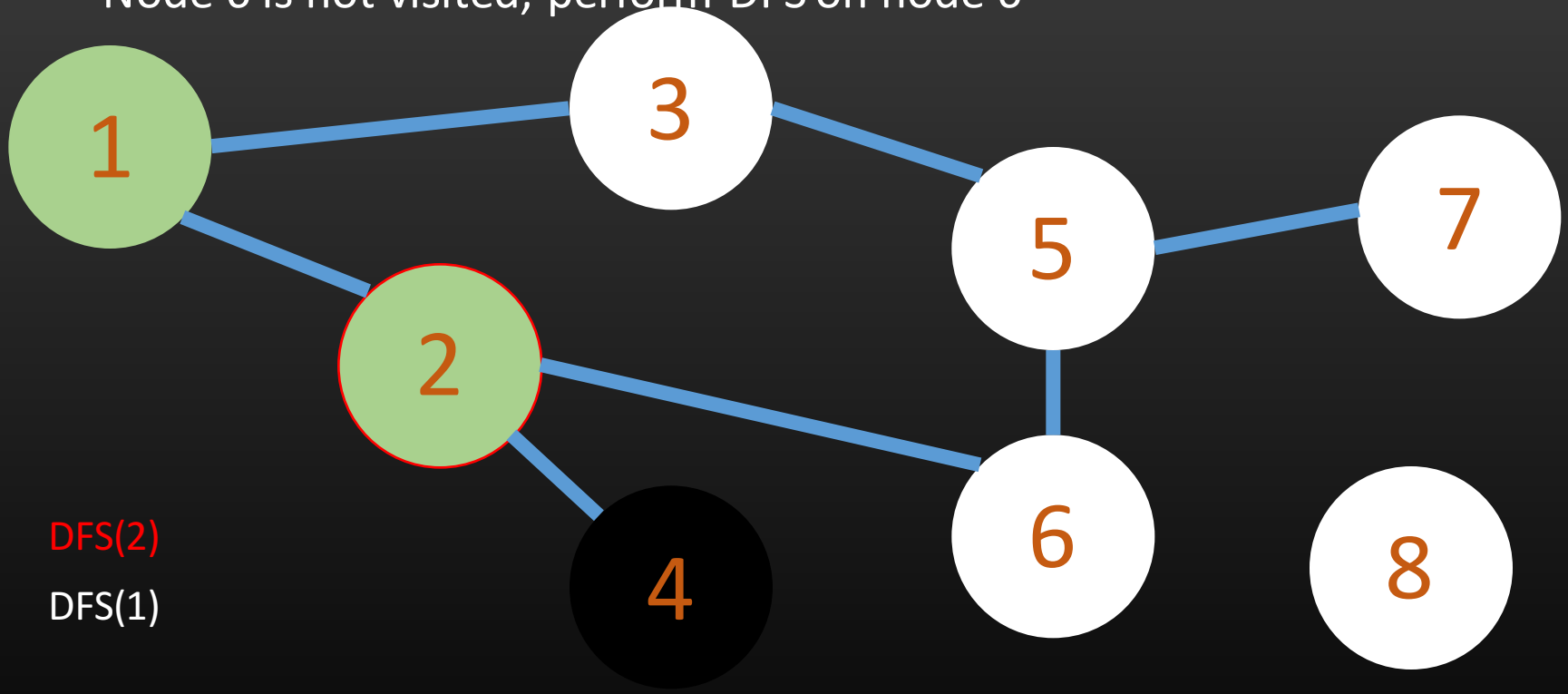


DFS(2)

DFS(1)

Depth-first Search - example

- Node 6 is not visited, perform DFS on node 6

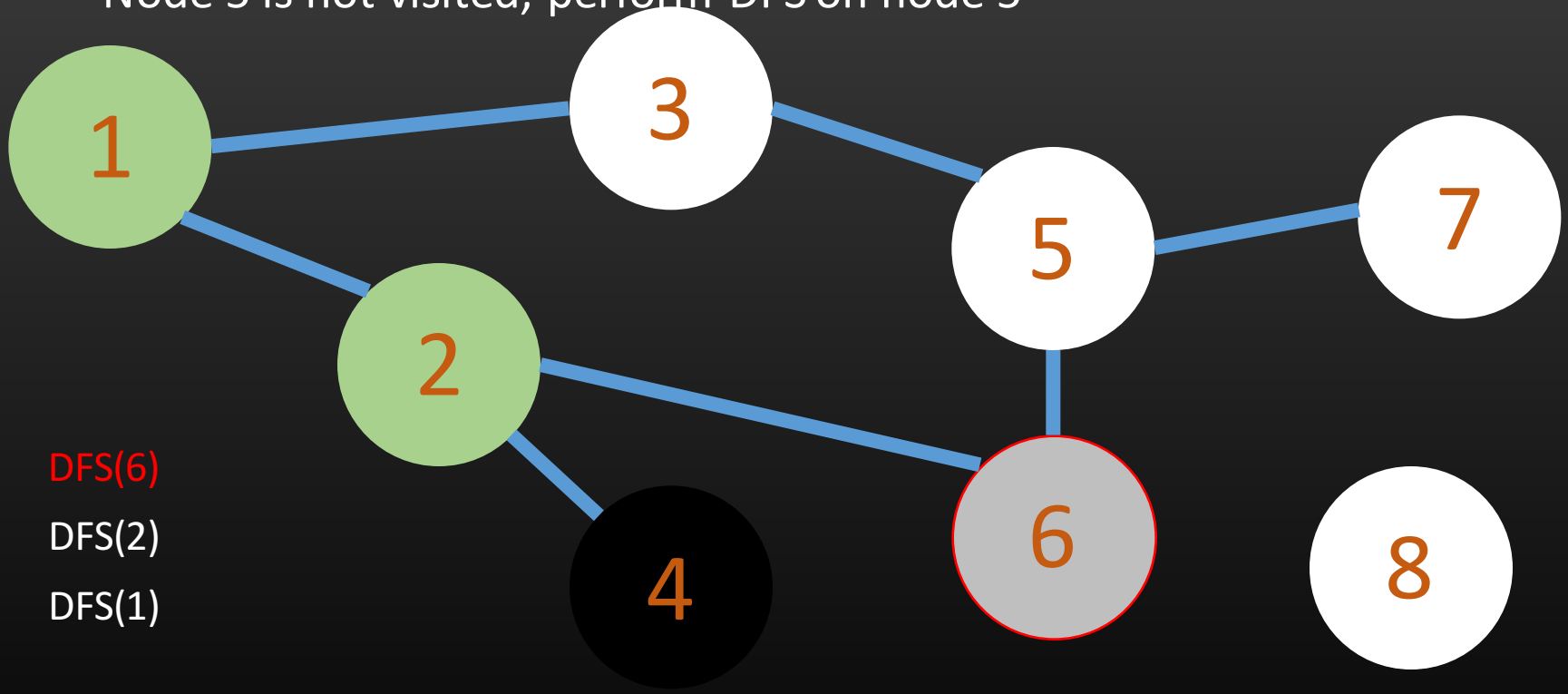


DFS(2)

DFS(1)

Depth-first Search - example

- Node 5 is not visited, perform DFS on node 5



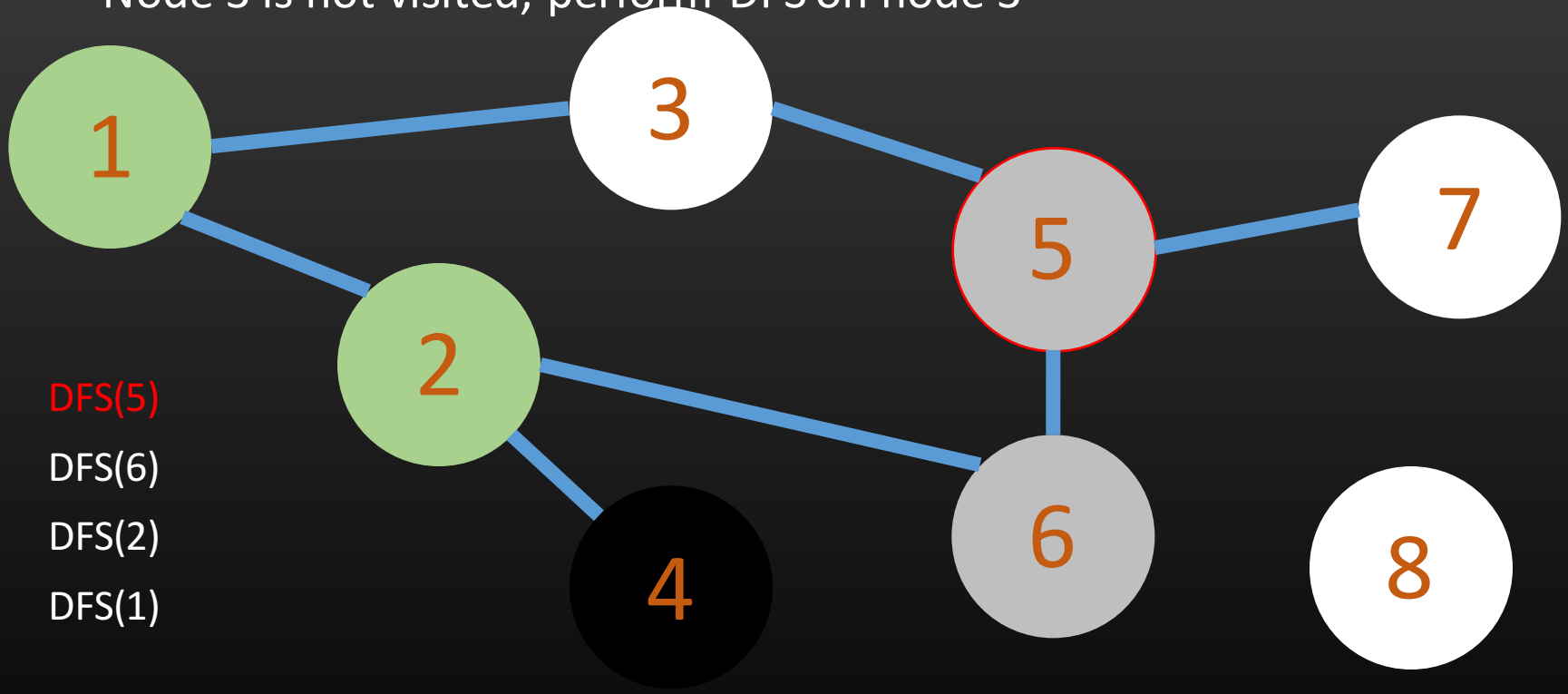
DFS(6)

DFS(2)

DFS(1)

Depth-first Search - example

- Node 3 is not visited, perform DFS on node 3



DFS(5)

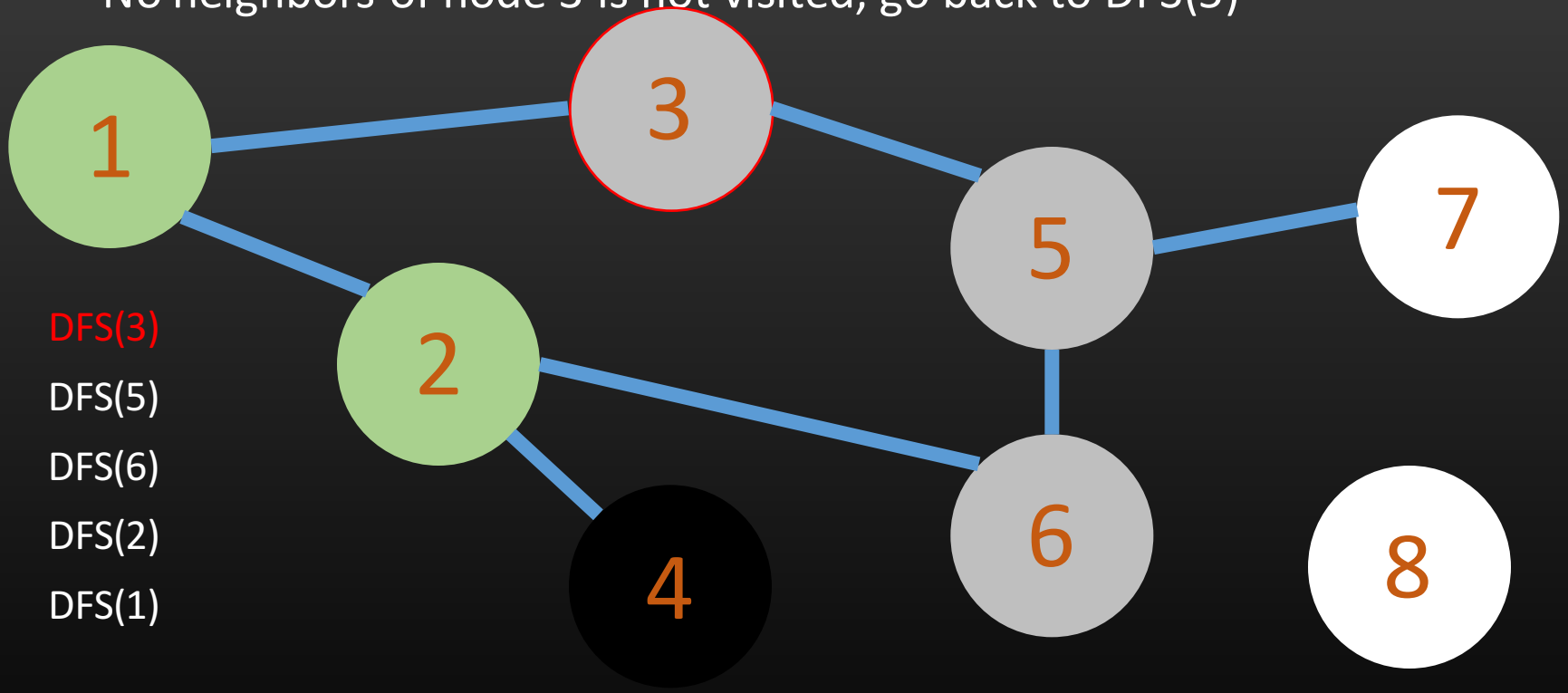
DFS(6)

DFS(2)

DFS(1)

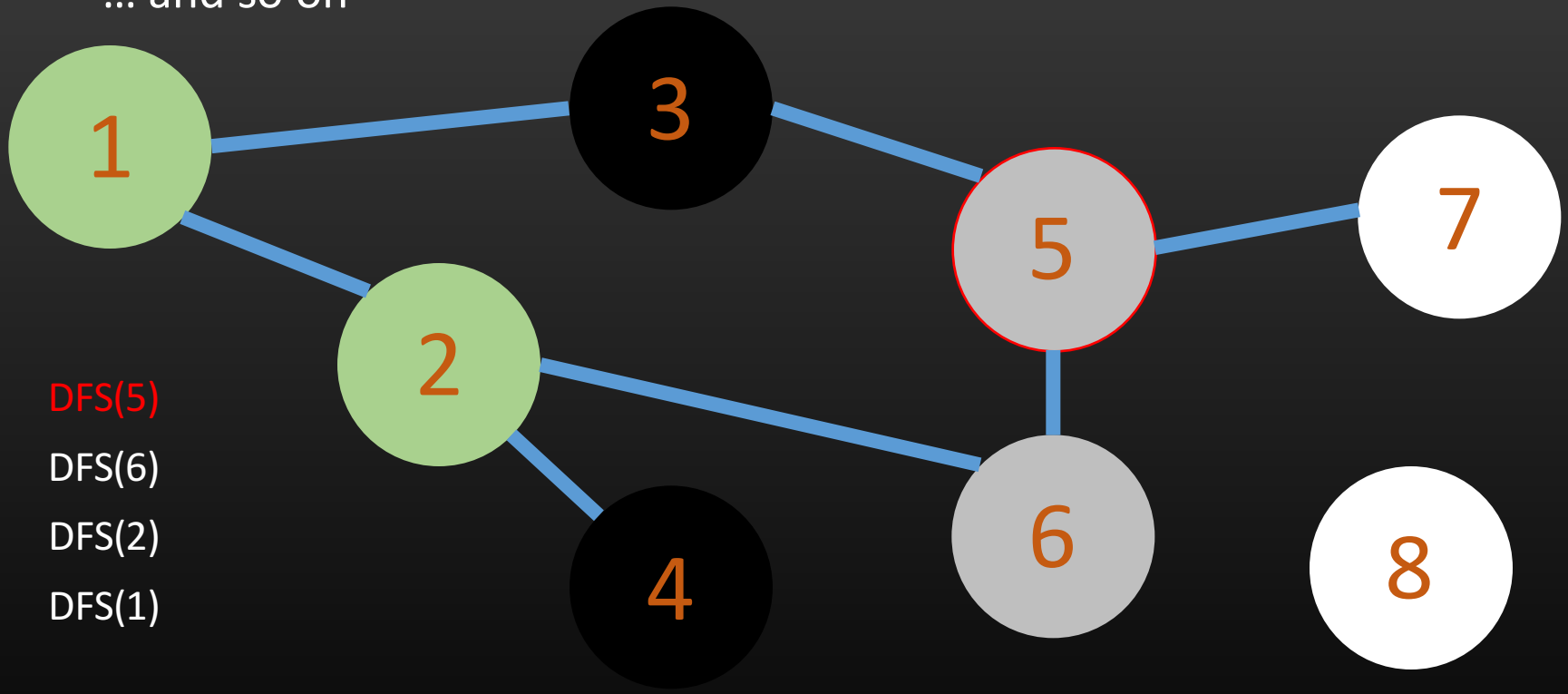
Depth-first Search - example

- No neighbors of node 3 is not visited, go back to DFS(5)



Depth-first Search - example

- … and so on



DFS(5)

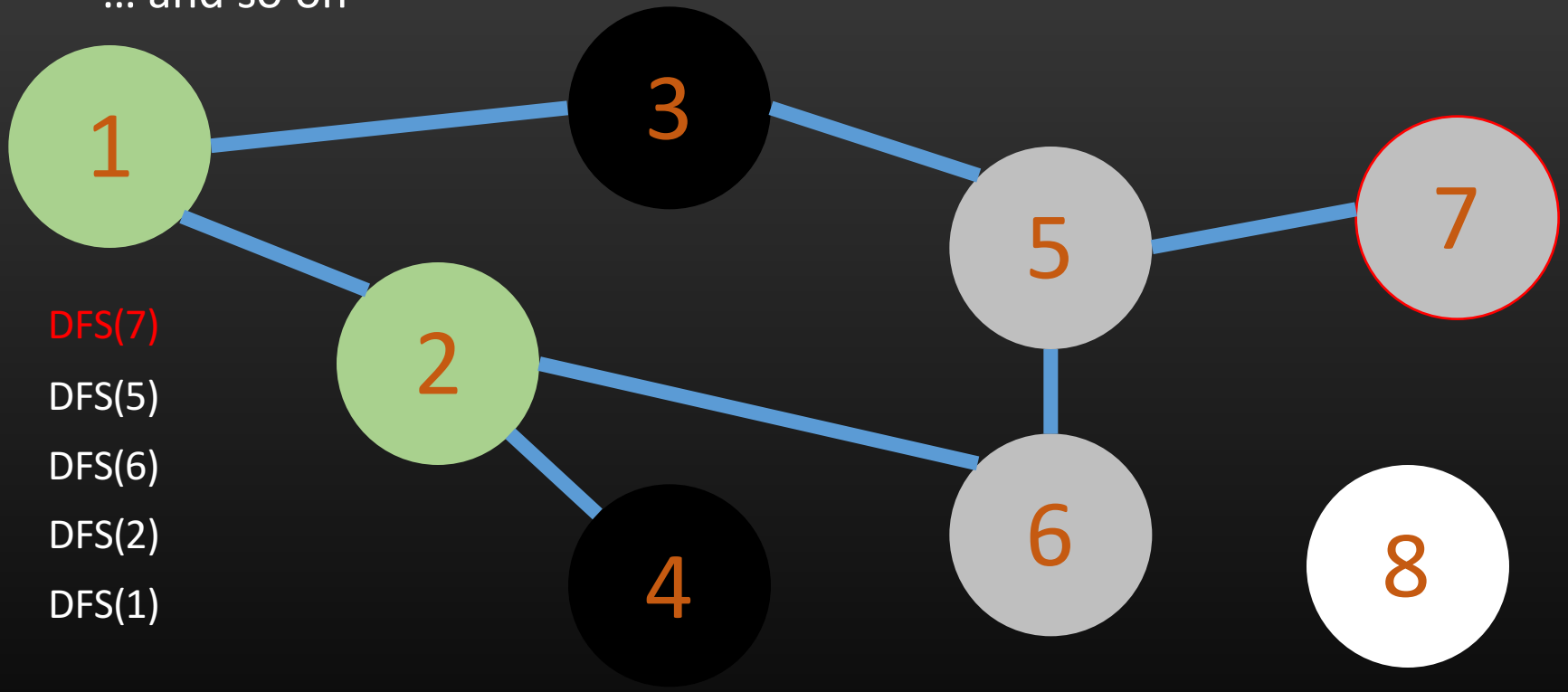
DFS(6)

DFS(2)

DFS(1)

Depth-first Search - example

- … and so on



DFS(7)

DFS(5)

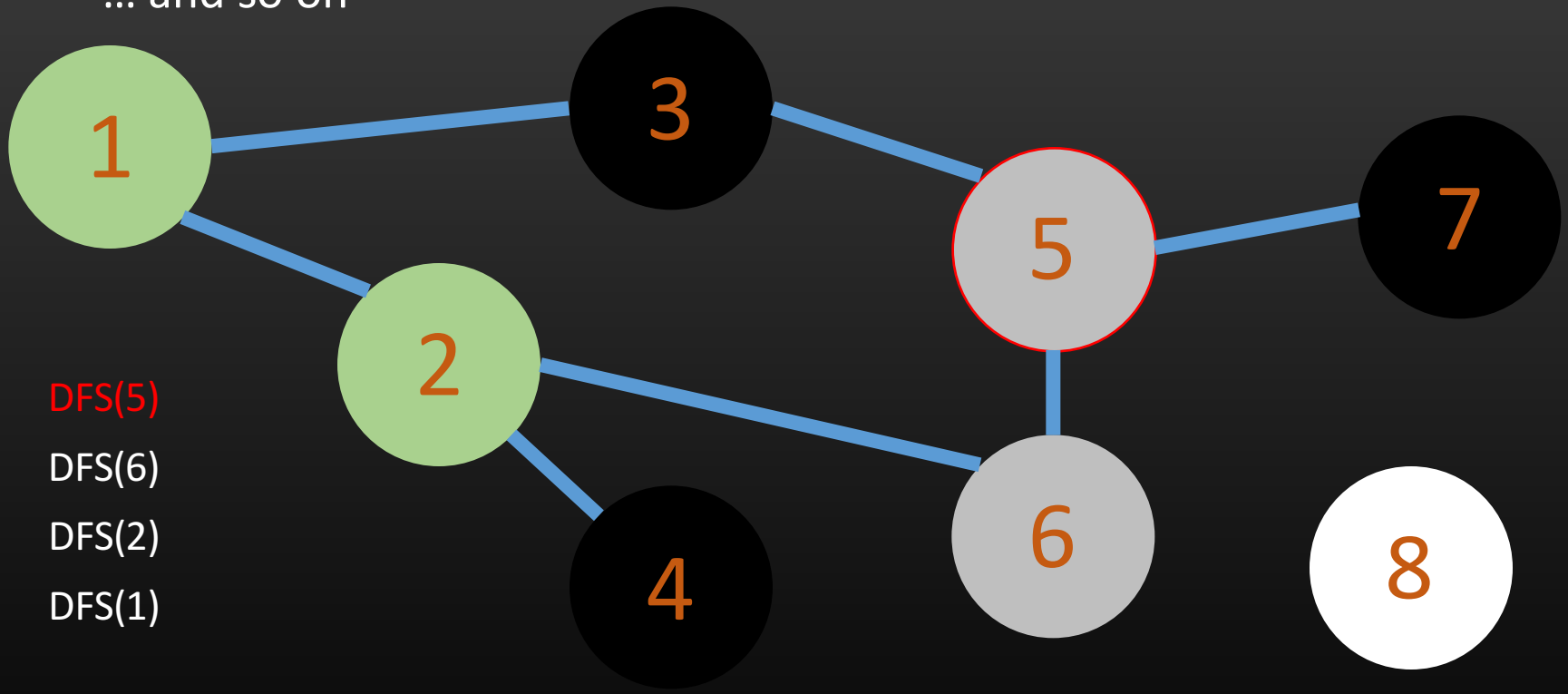
DFS(6)

DFS(2)

DFS(1)

Depth-first Search - example

- … and so on



DFS(5)

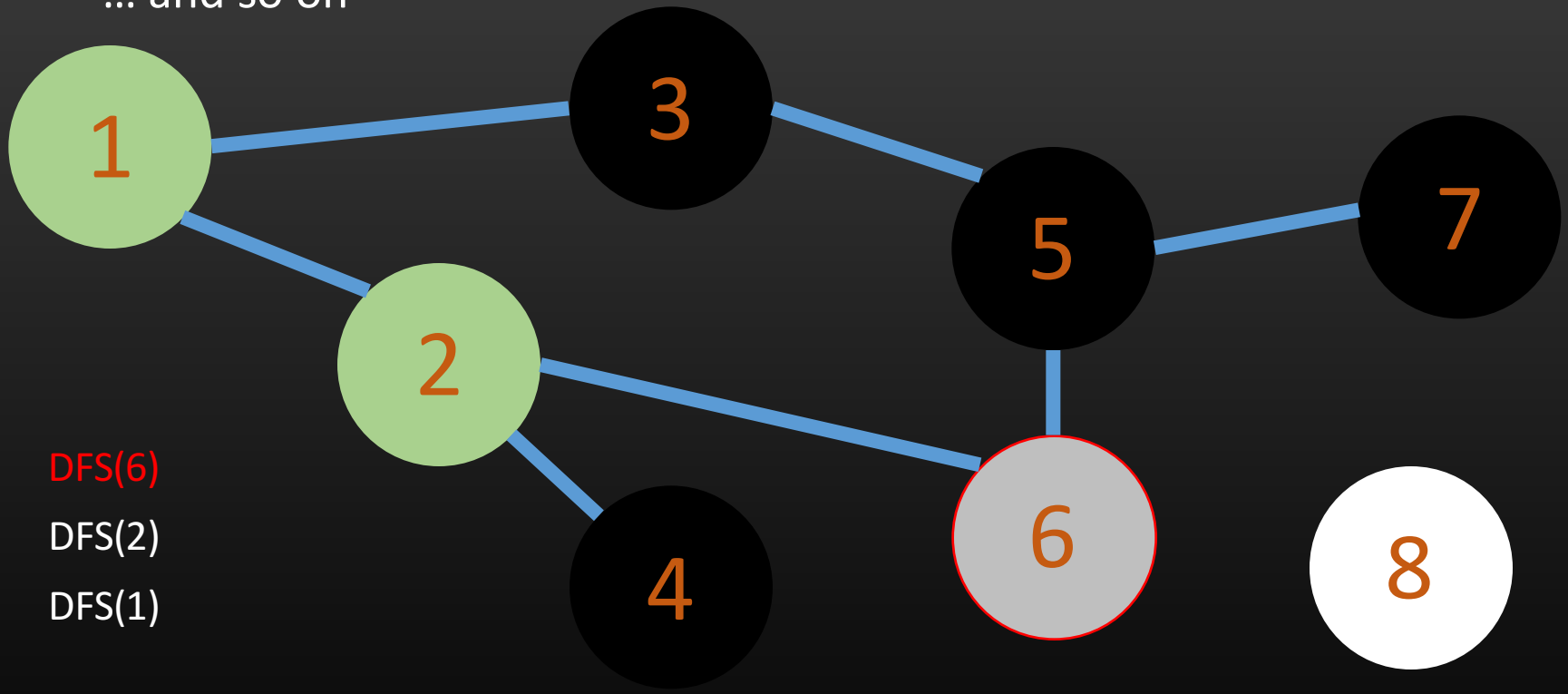
DFS(6)

DFS(2)

DFS(1)

Depth-first Search - example

- … and so on



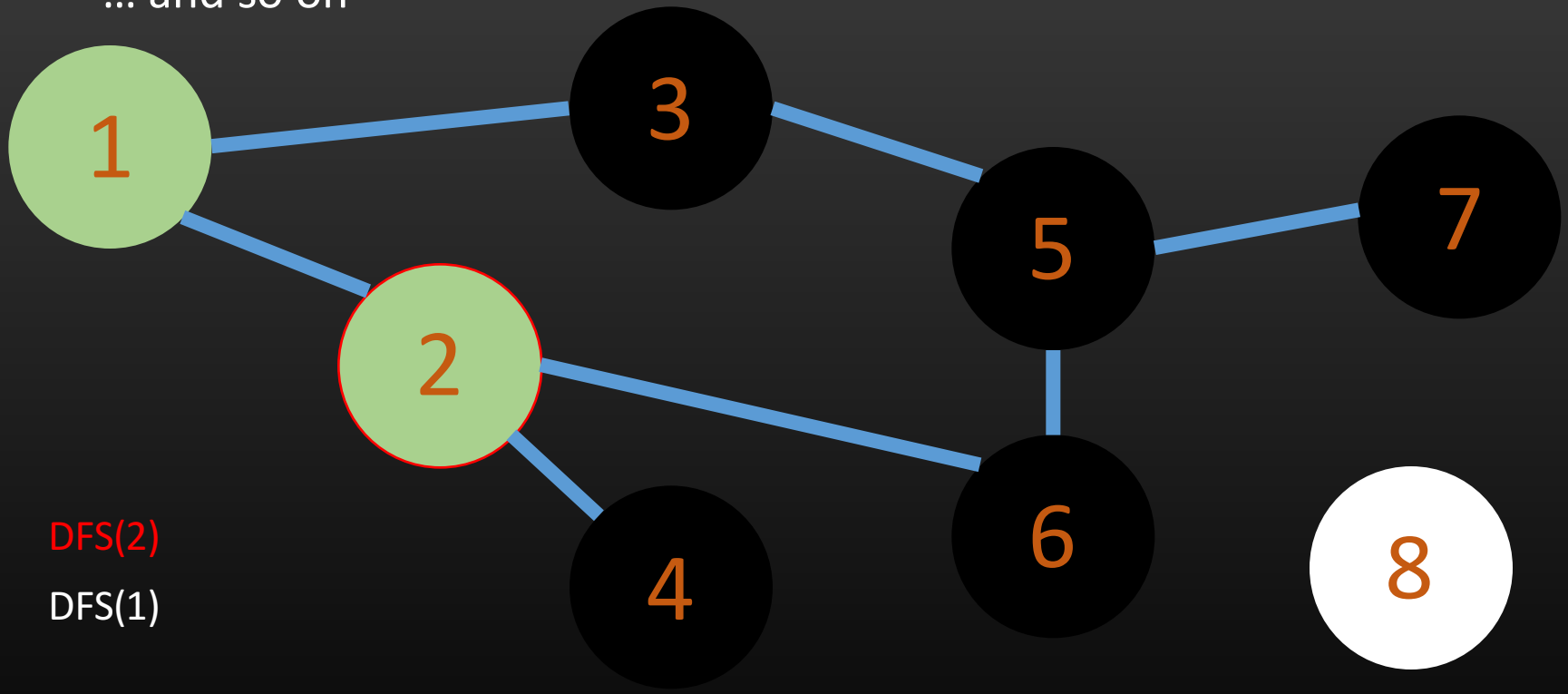
DFS(6)

DFS(2)

DFS(1)

Depth-first Search - example

- … and so on

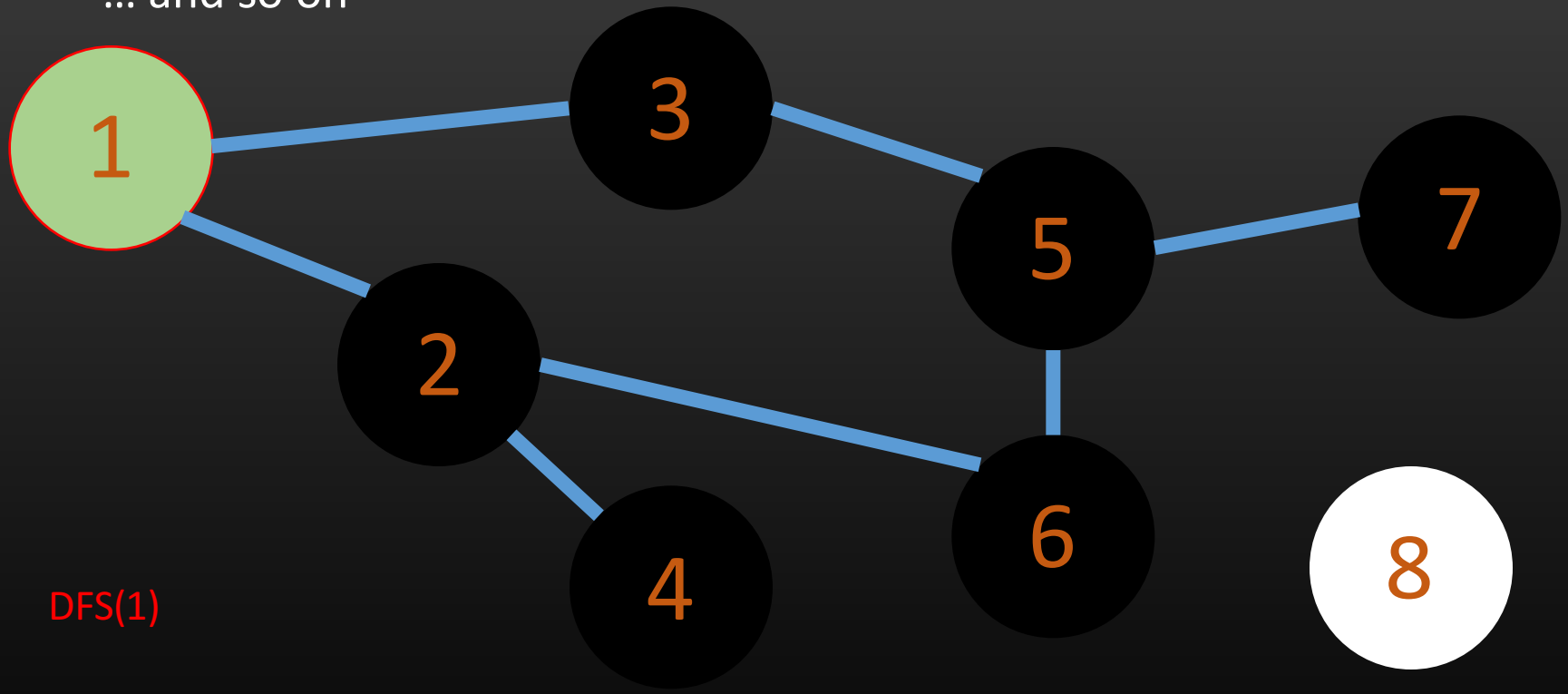


DFS(2)

DFS(1)

Depth-first Search - example

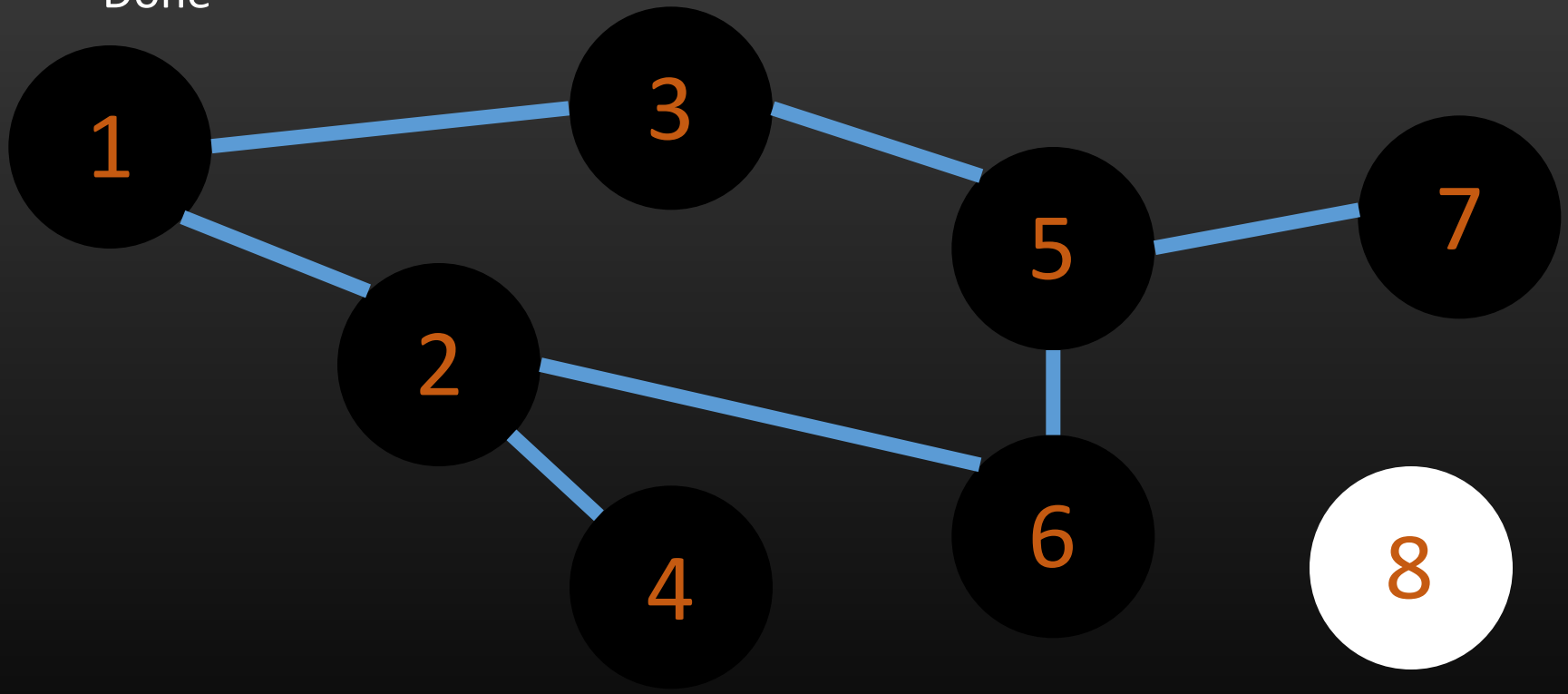
- … and so on



DFS(1)

Depth-first Search - example

- Done



Depth-first Search - Implementation

```
procedure DFS(vertex v){
    label v as discovered
    for all w adjacent to v do
        if w is not labeled as discovered then
            DFS(w)
    }
```

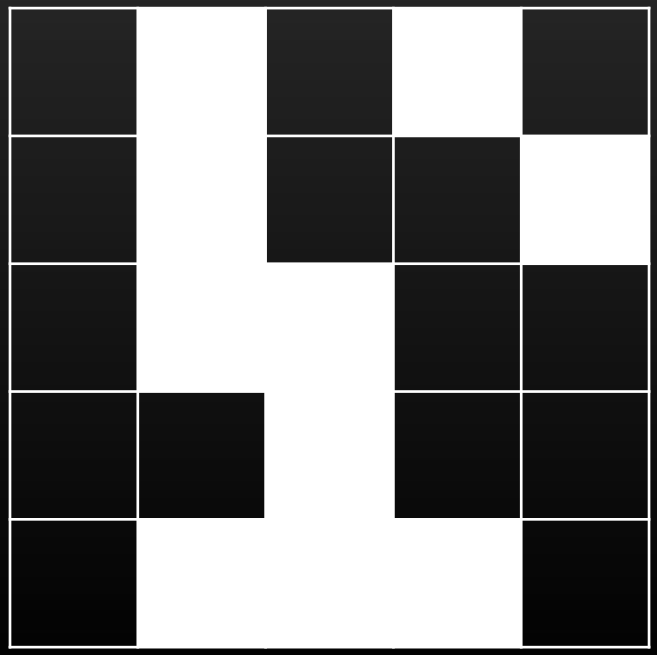
Depth-first Search

- Time Complexity:
  - $O(|V|^2)$ when using adjacency matrix
  - $O(|V|+|E|)$ when using adjacency list / edge list

Depth-first Search - Example

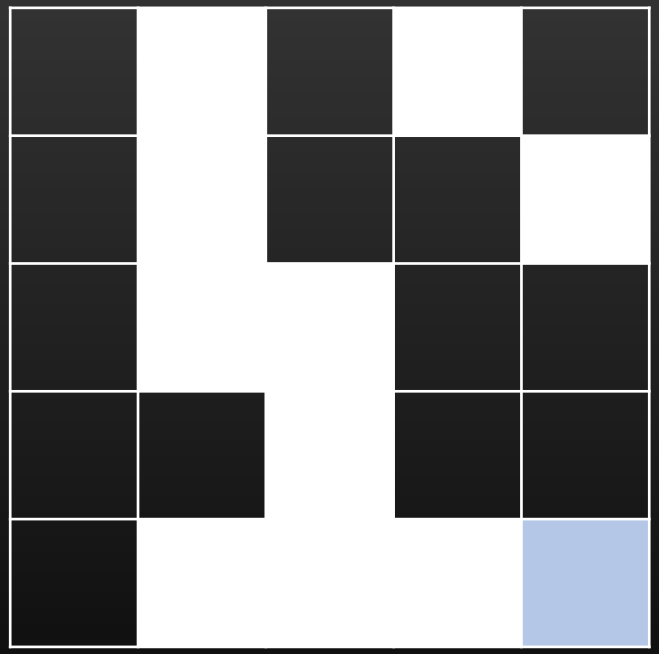
- Flood fill
- Determine the number of nodes (or area) connected to a given node

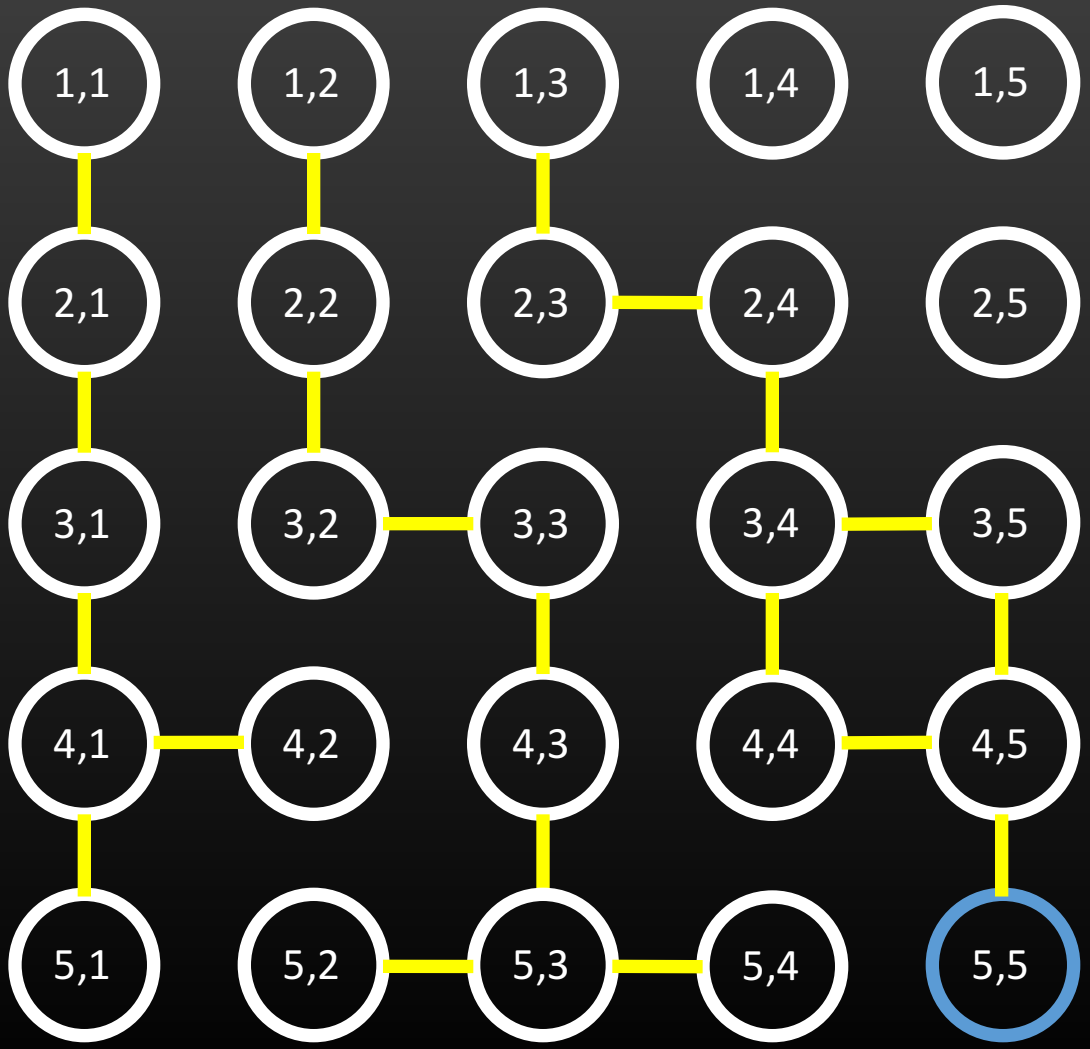
- E.g. Find the area connected to the cell (5, 5) with the same color in the following grid



Flood fill

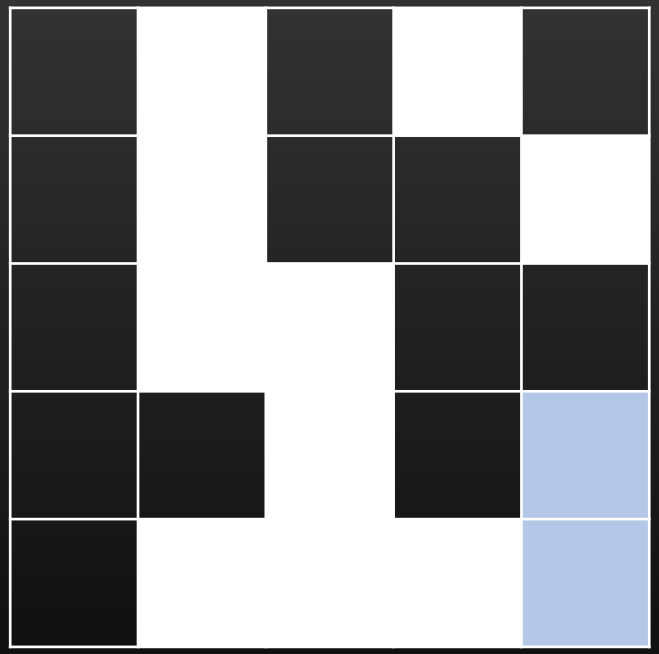
- Perform DFS on (5, 5)

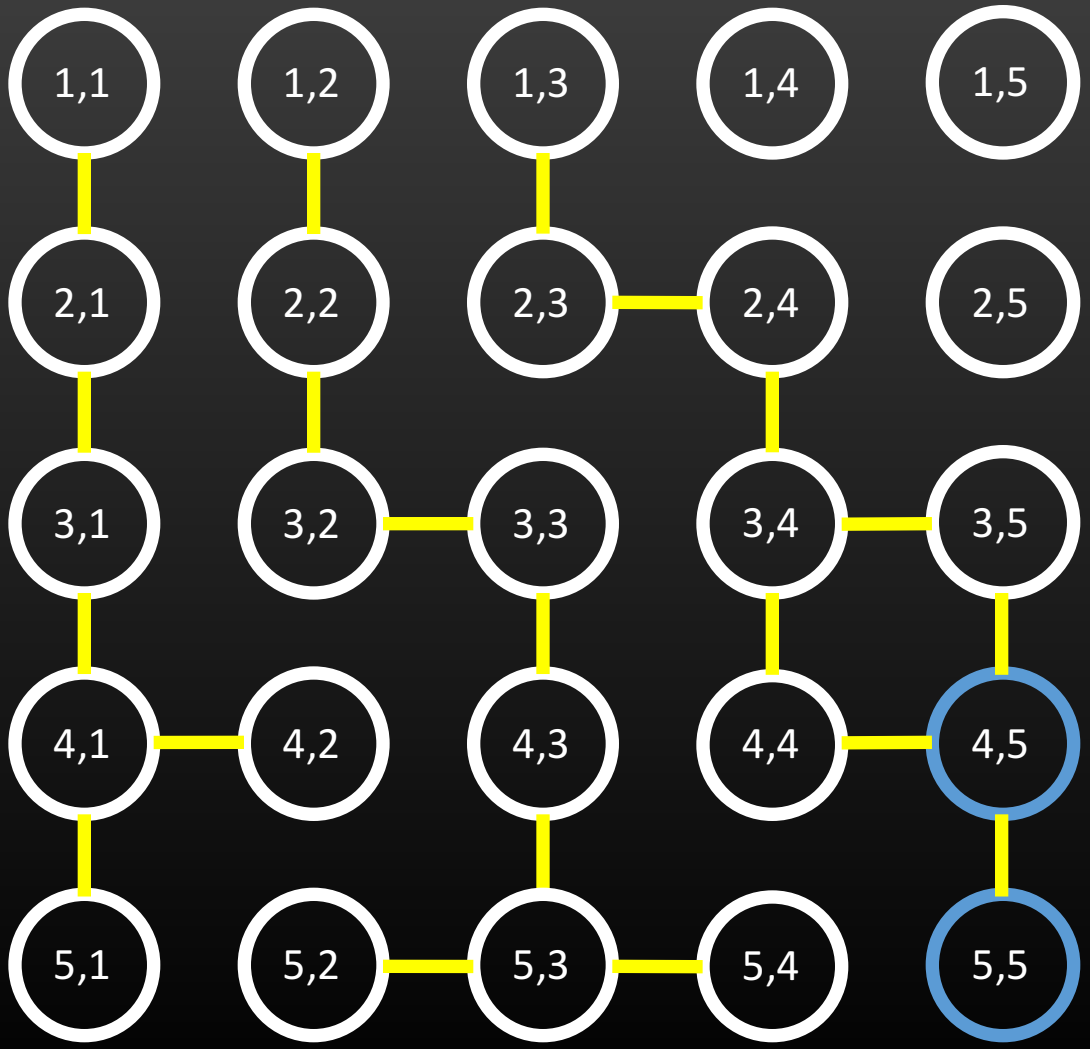




Flood fill

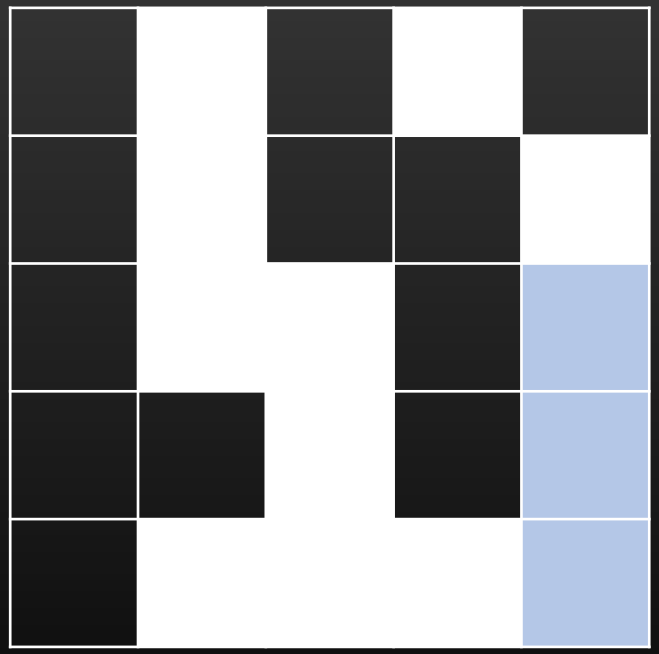
- Perform DFS on (4, 5)

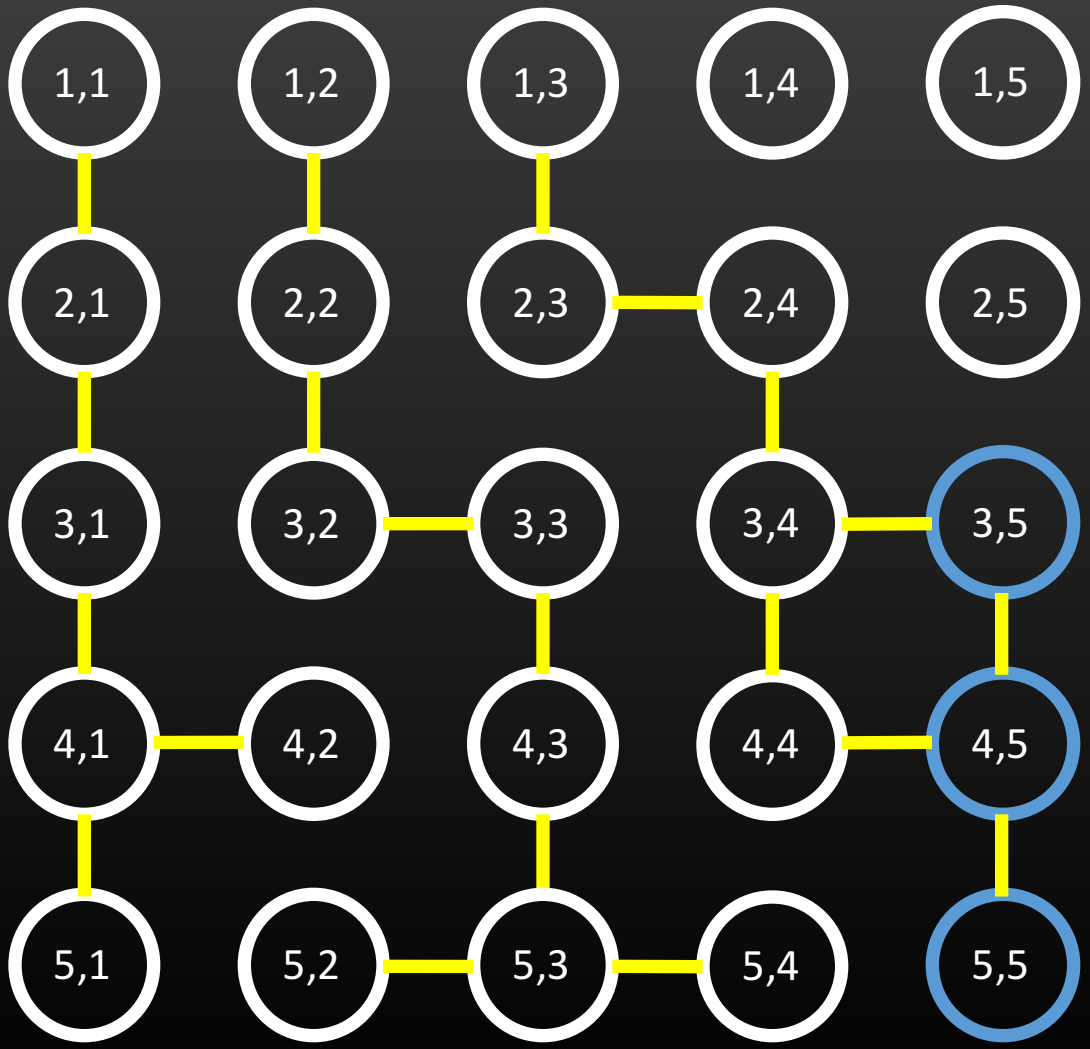




Flood fill

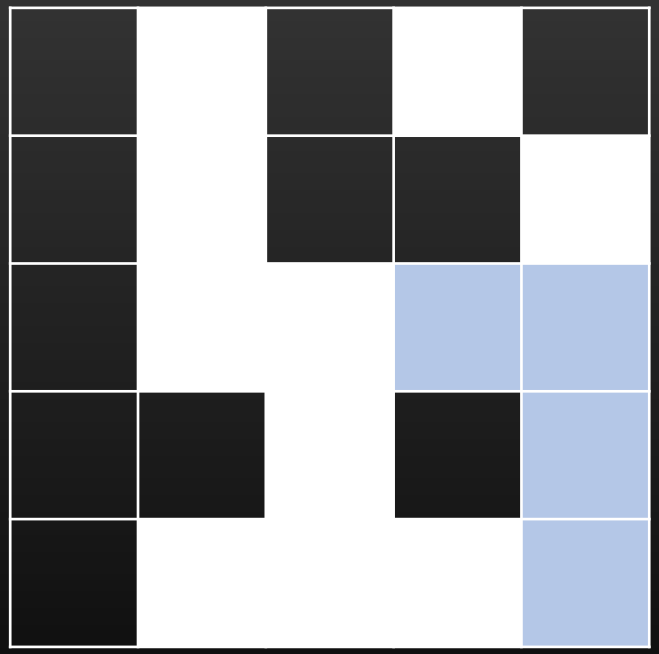
- Perform DFS on (3, 5)

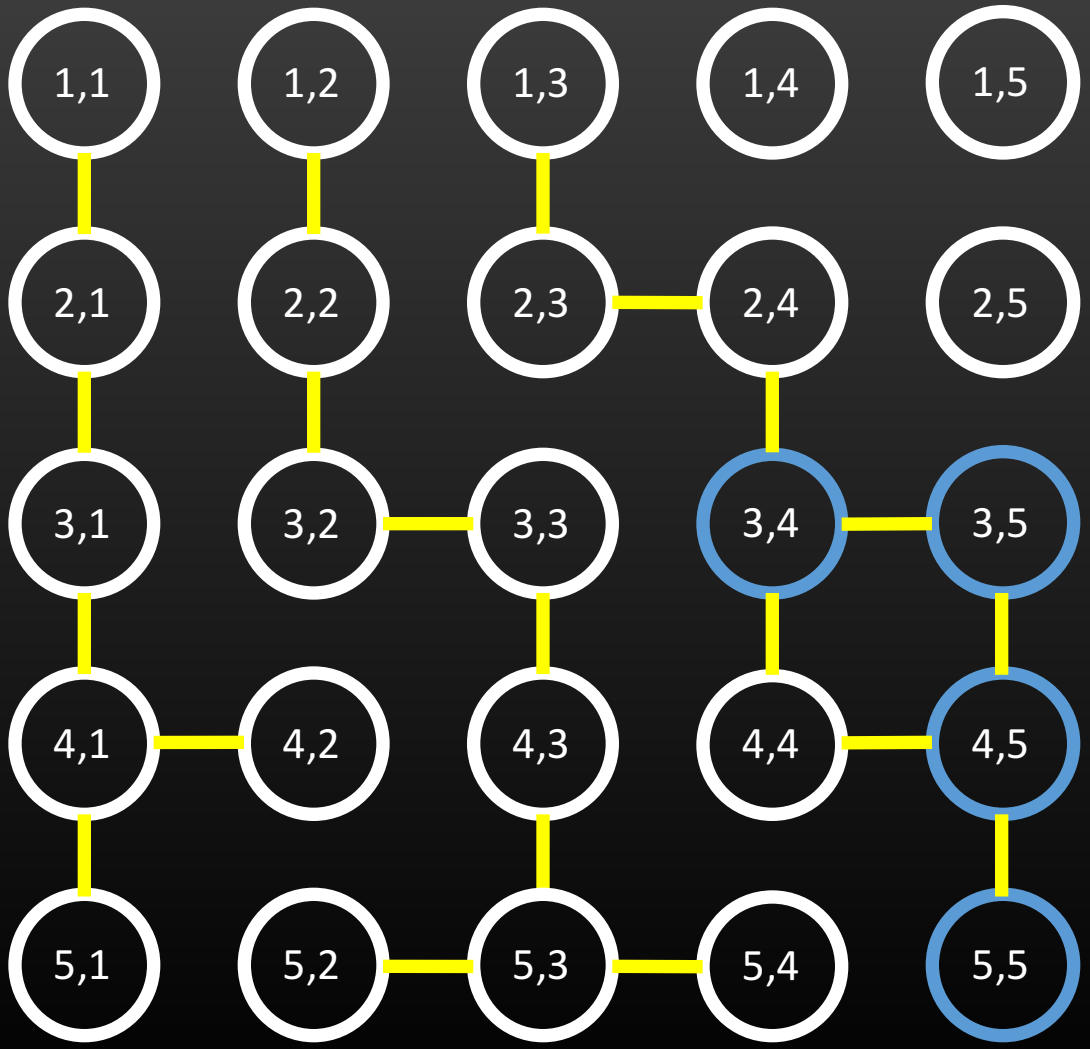




Flood fill

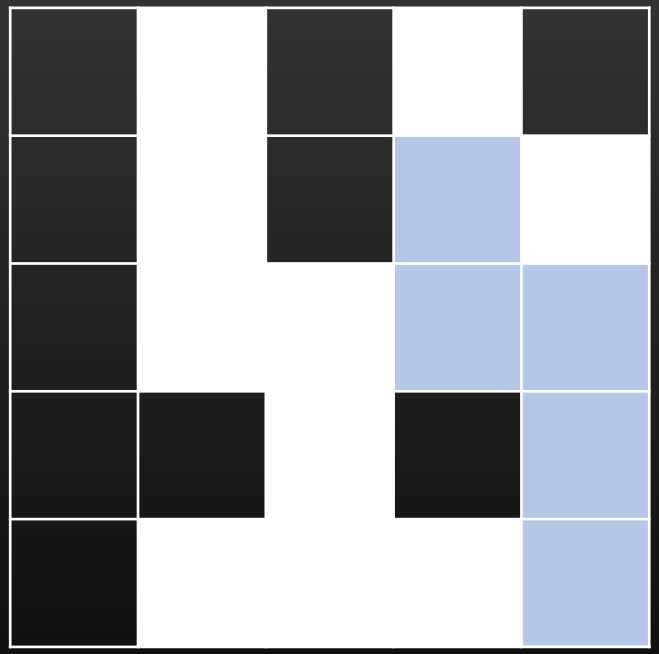
- … and so on

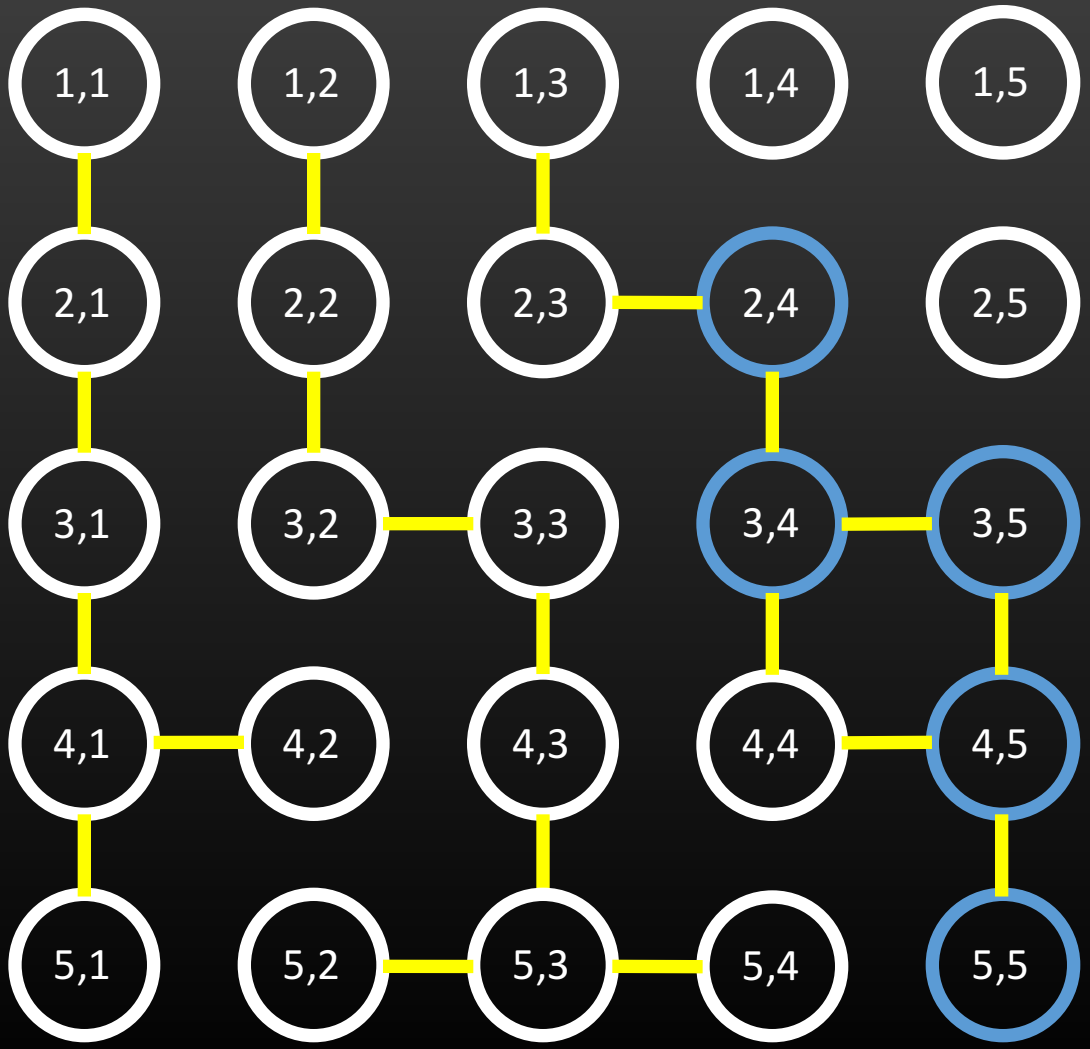




Flood fill

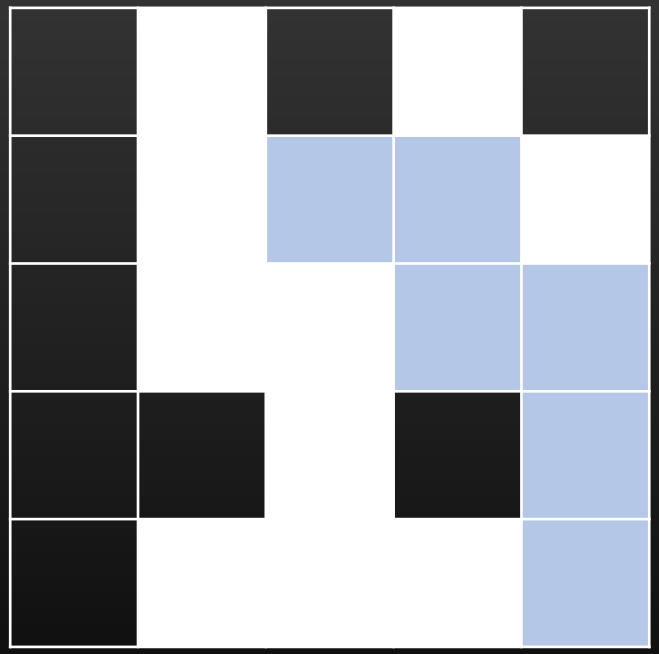
- … and so on

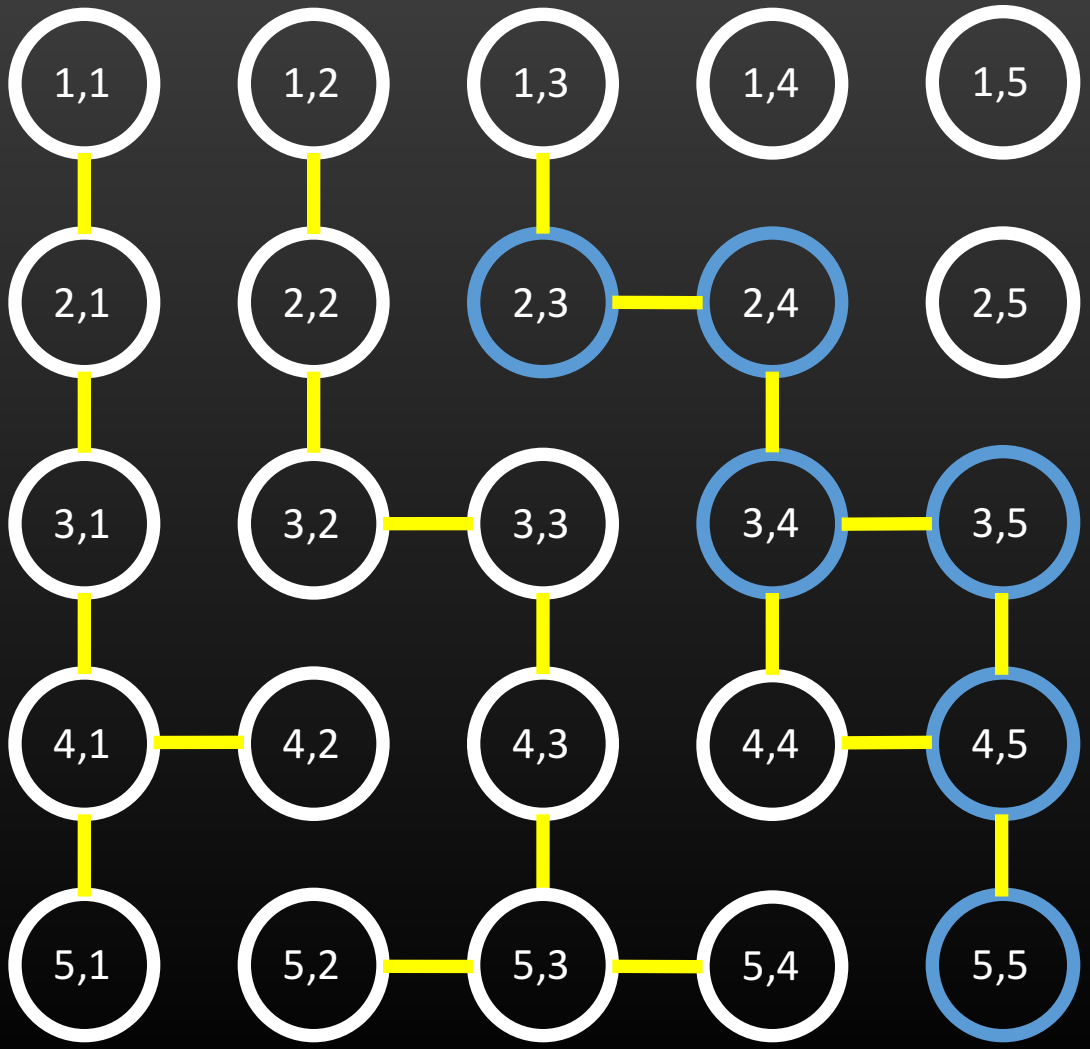




Flood fill

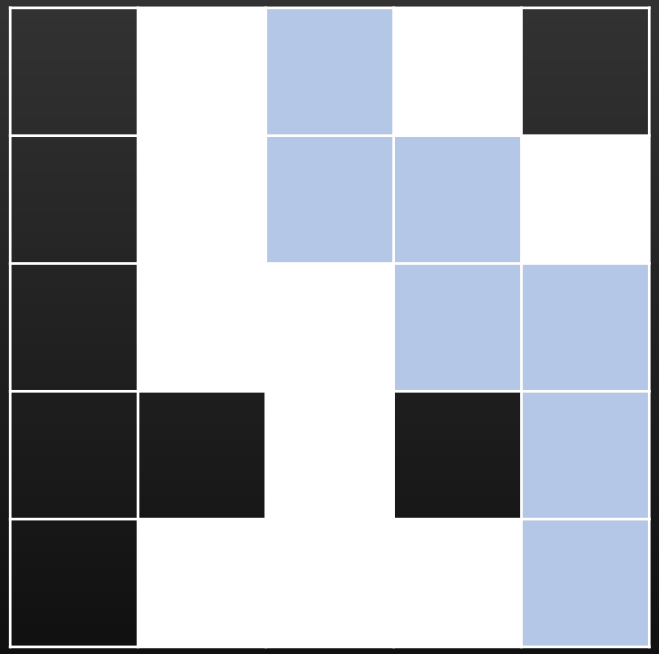
- … and so on

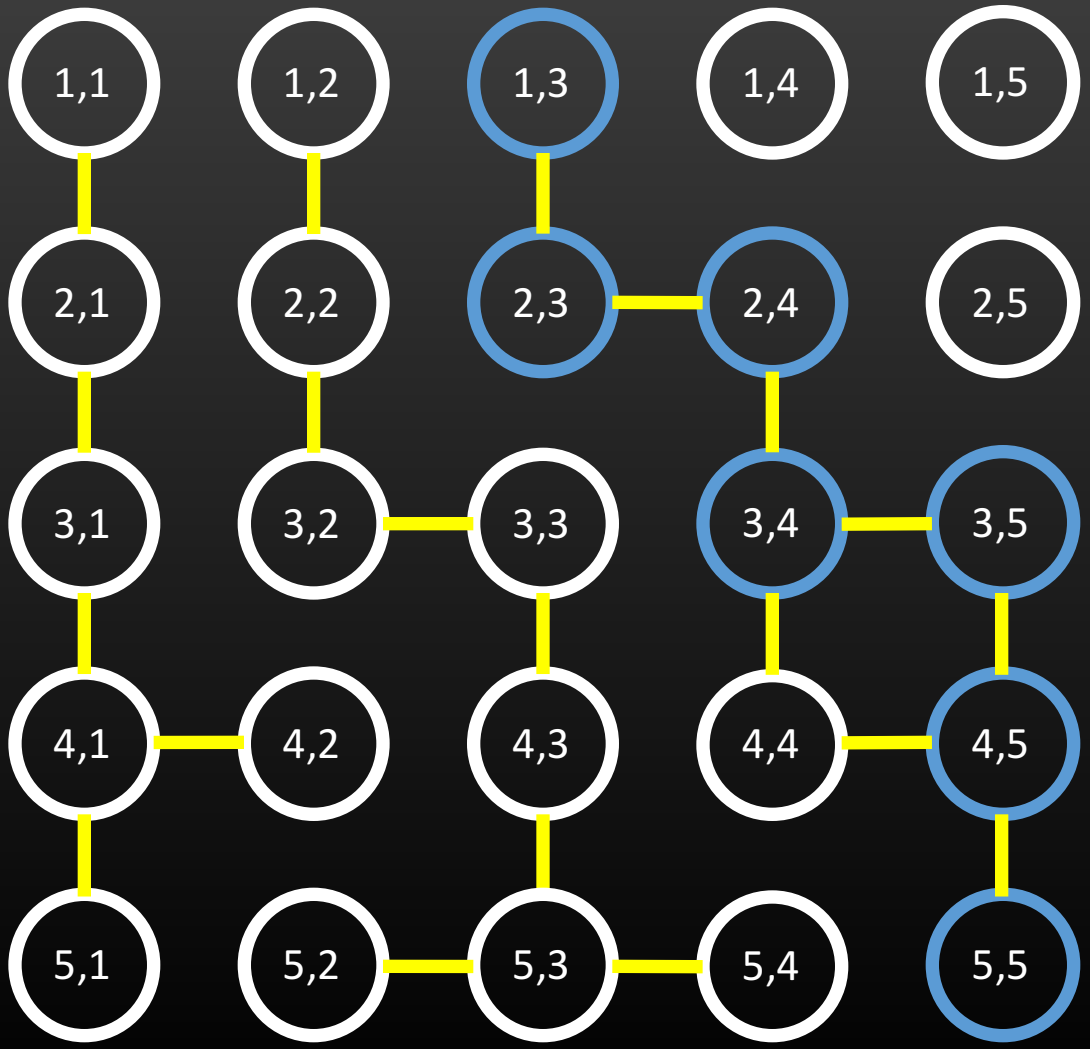




Flood fill

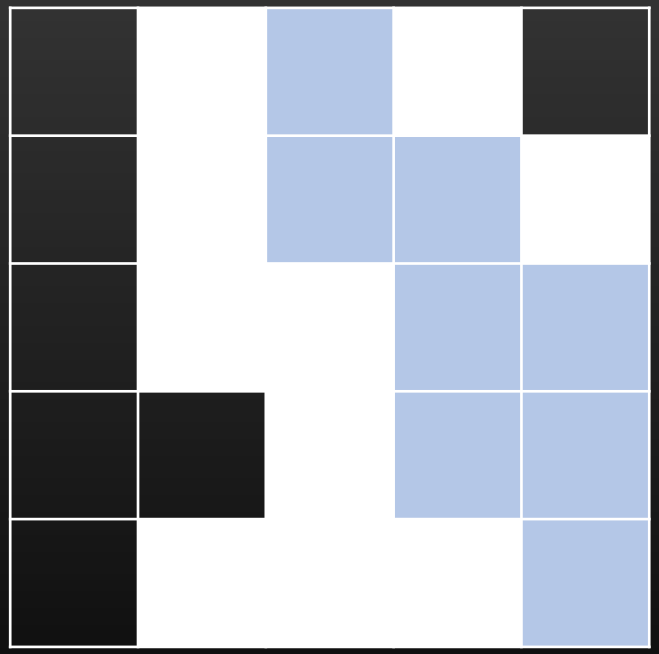
- … and so on

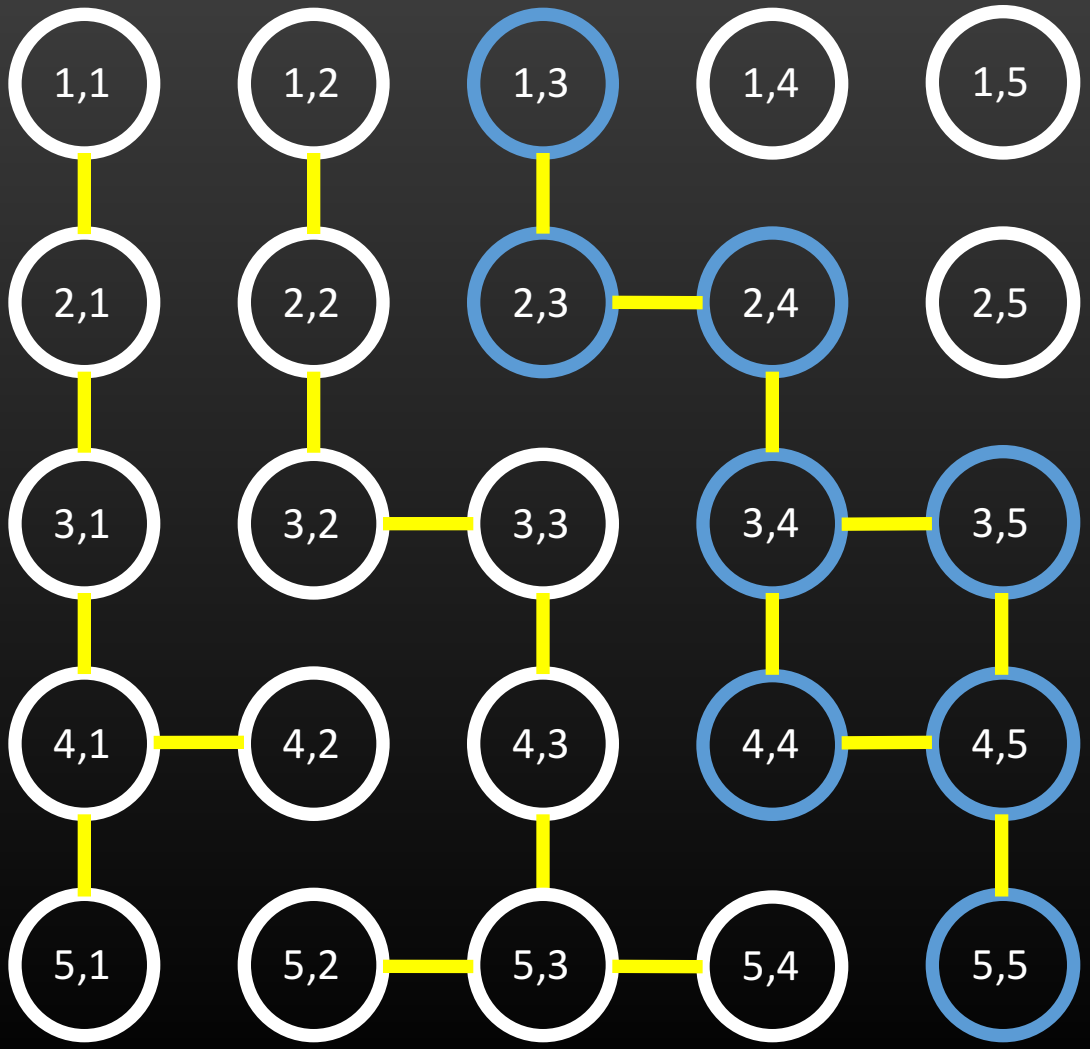




Flood fill

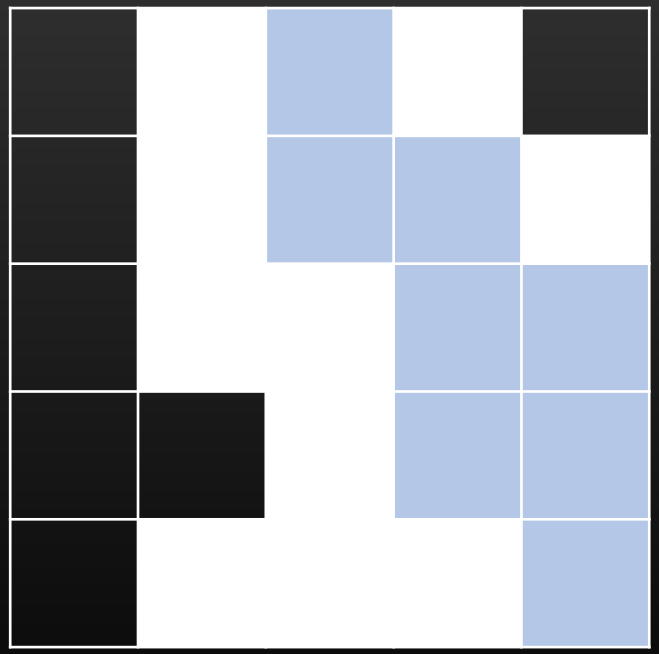
- … and so on

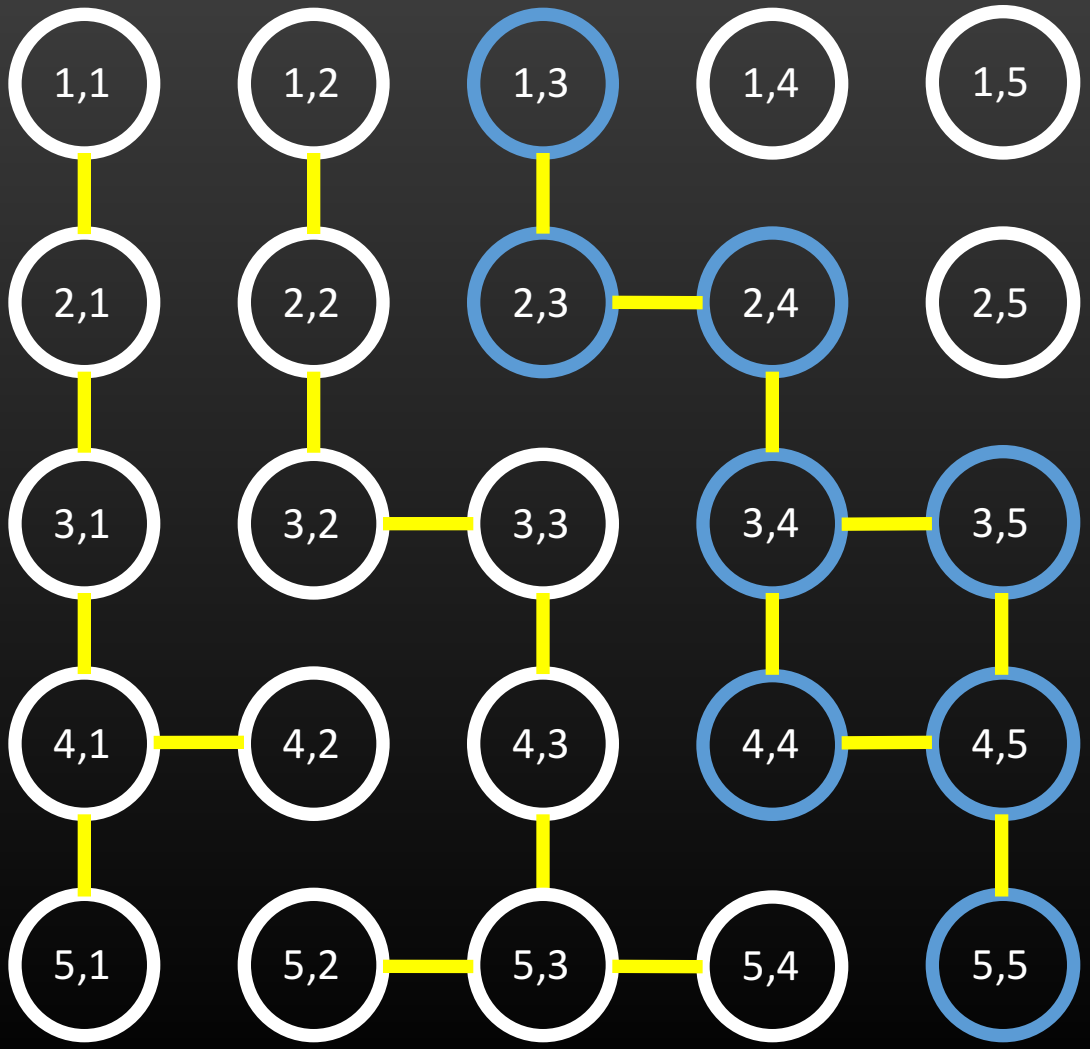




Flood fill

- Count the # of visited nodes during DFS





Flood fill - Implementation

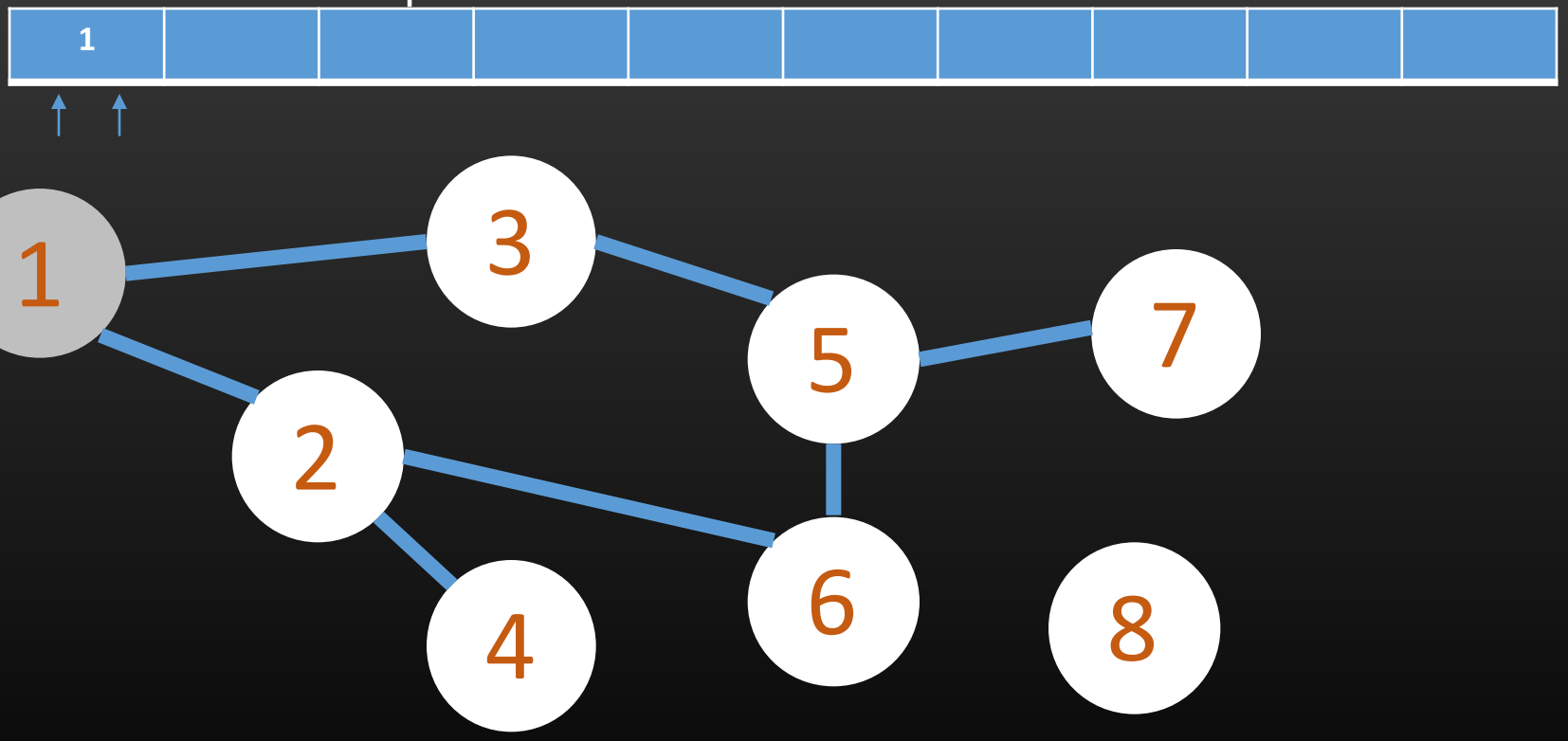
```
Procedure floodFill(x, y){
        label node(x, y) as discovered
        counter = counter + 1
        For each node(newX, newY) in neighbors of node(x, y) do
                If node(newX, newY) is valid and not discovered then
                        floodFill(newX, newY)
```

Breadth-first Search

- Starts at a node
- Explores the neighbor nodes first before moving to the next level neighbors
- The vertices closest to the source are evaluated first, and the most distant vertices are evaluated last
- Use a queue to implement

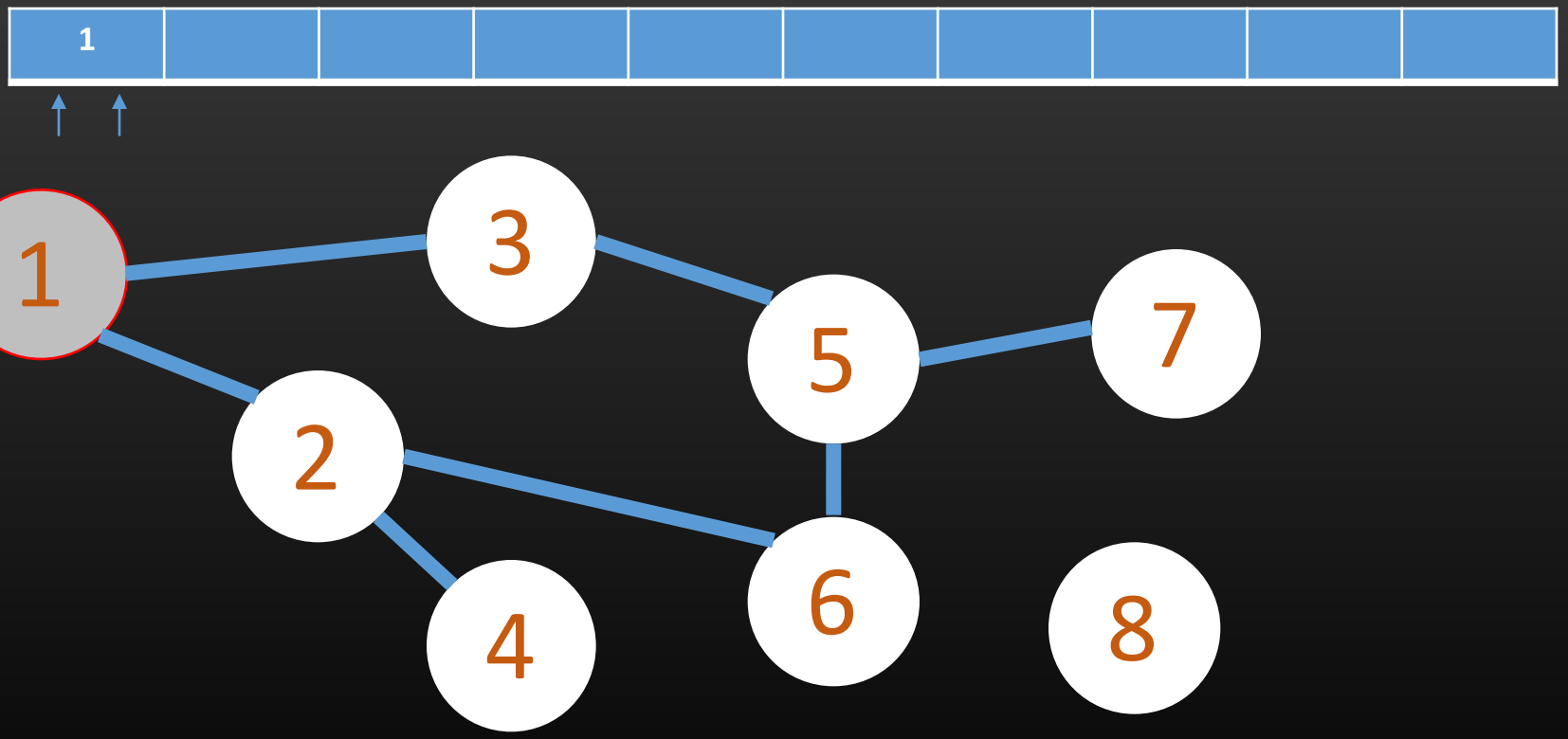
Breadth-first Search - example

- Push 1 into the queue and mark node 1 as visited first



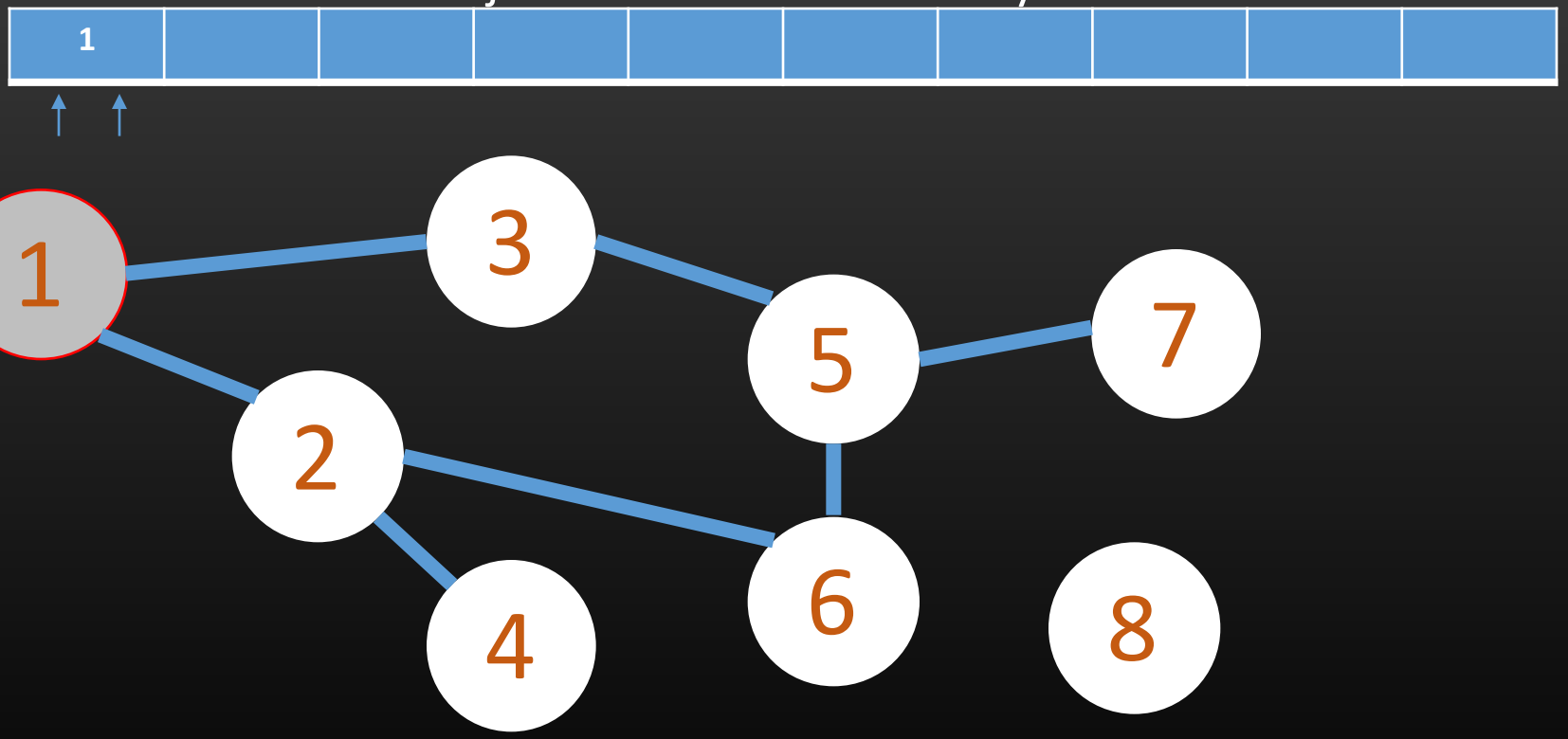
Breadth-first Search - example

- Perform BFS on node 1



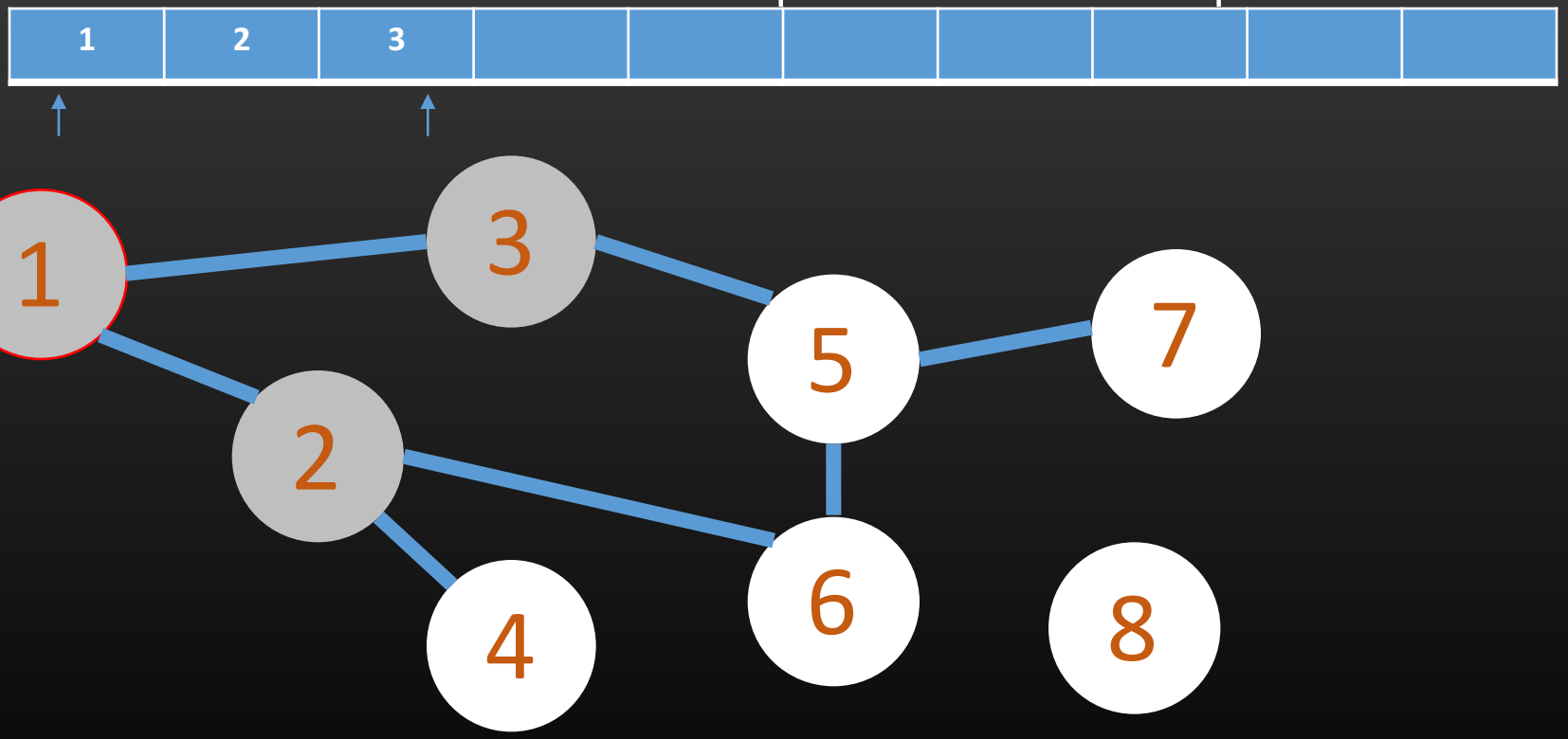
Breadth-first Search - example

- Node 2 and 3 are adjacent to node 1 and they are not visited



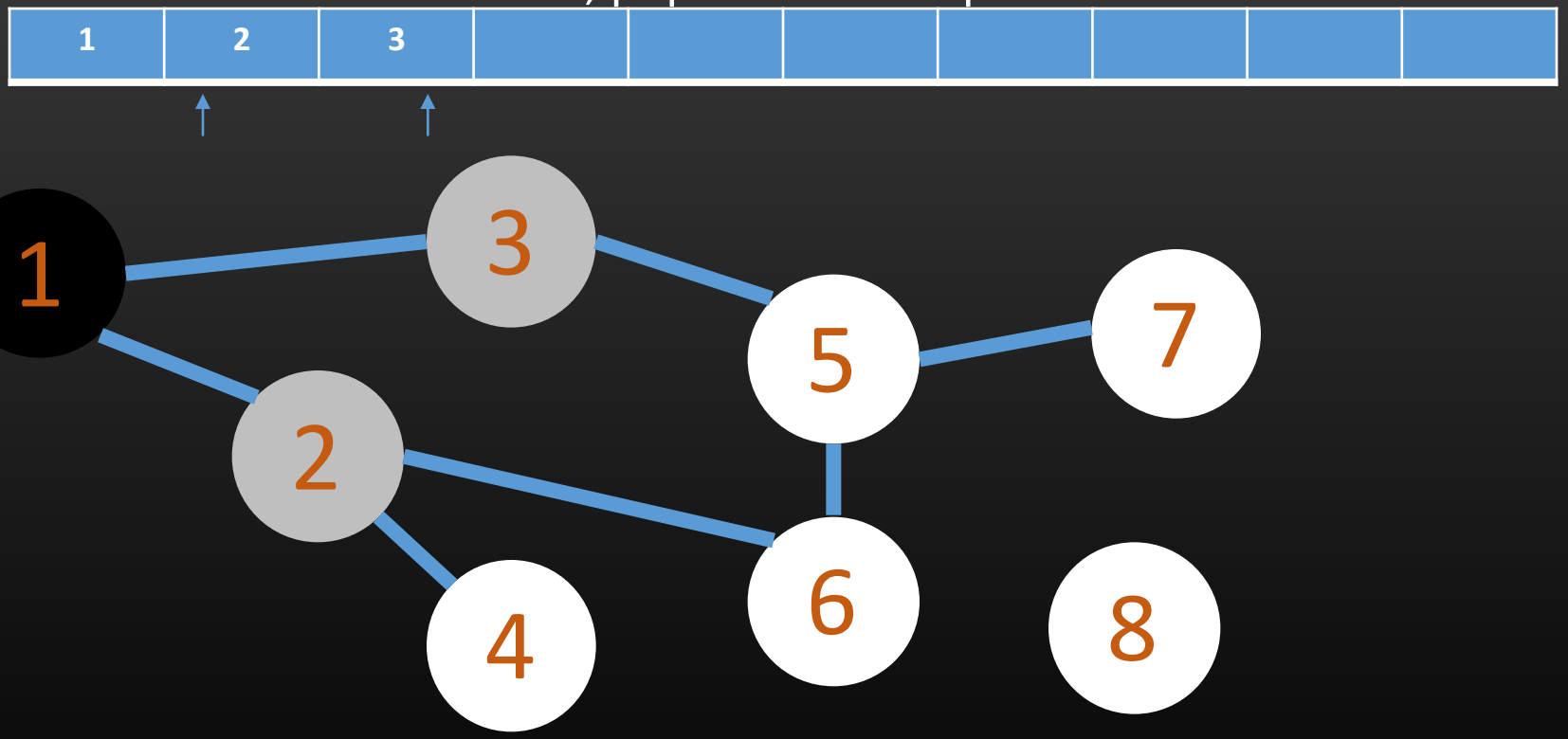
Breadth-first Search - example

- Mark node 2 and 3 as visited and push them into the queue



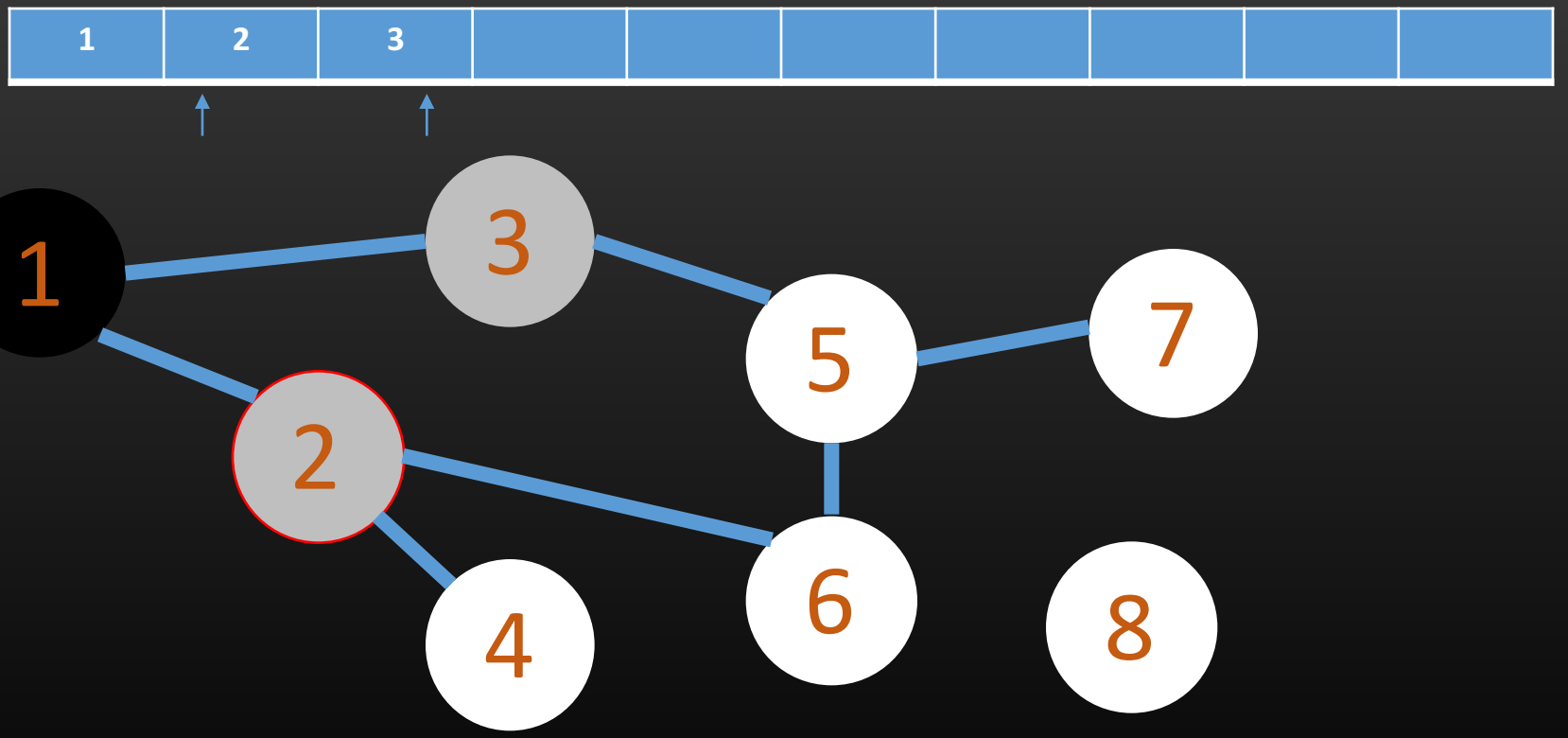
Breadth-first Search - example

- BFS on node 1 is finished, pop 1 from the queue



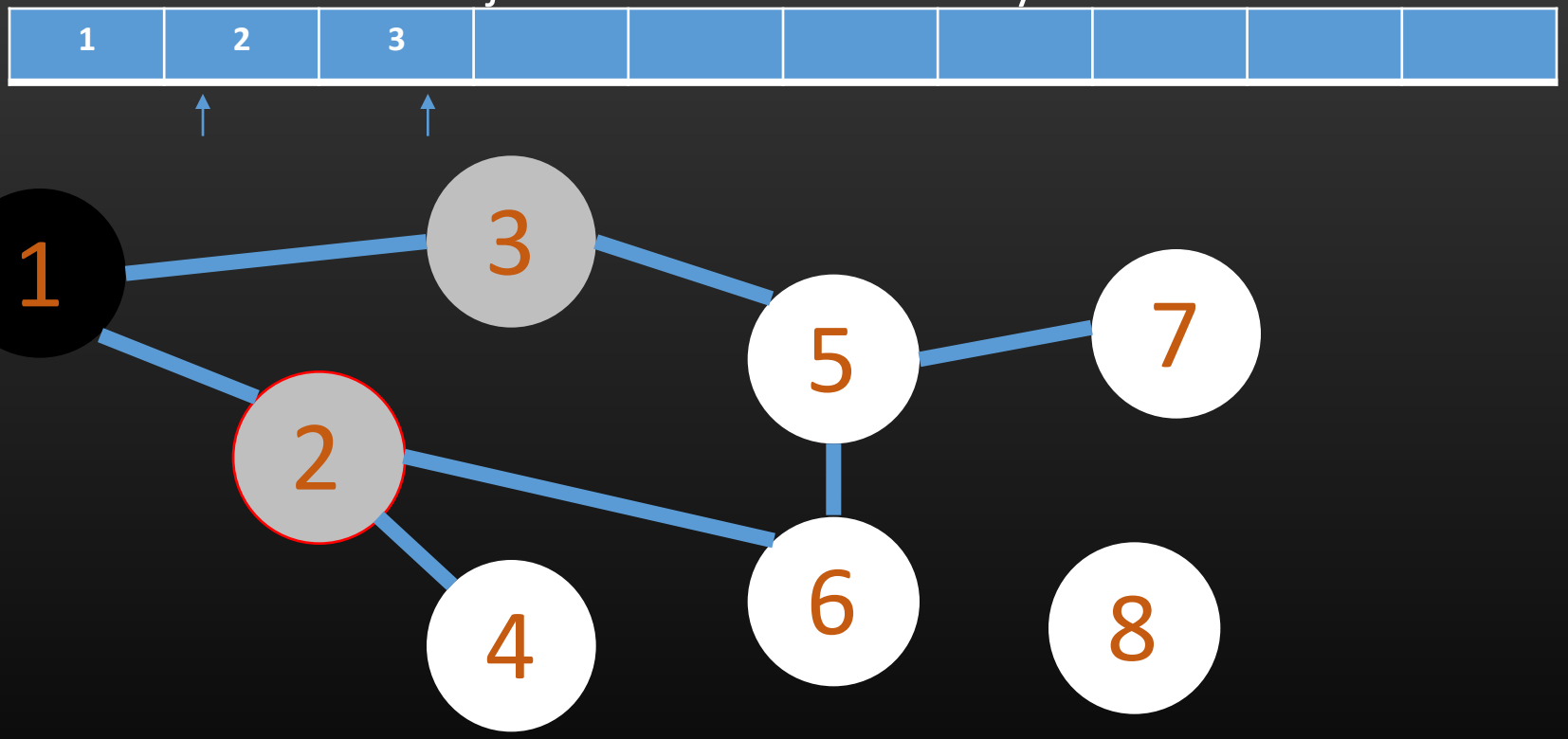
Breadth-first Search - example

- Perform BFS on node 2



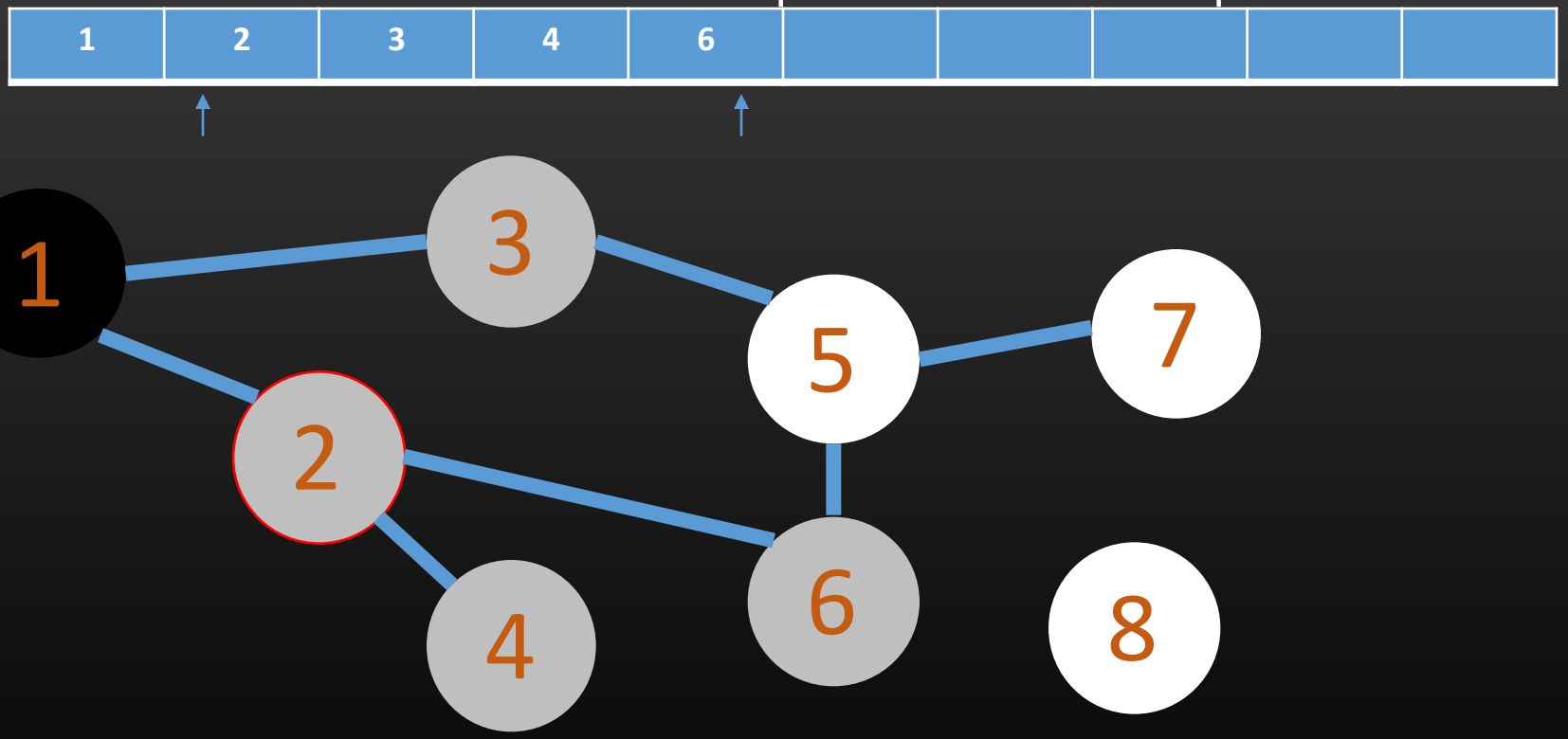
Breadth-first Search - example

- Node 4 and 6 are adjacent to node 2 and they are not visited



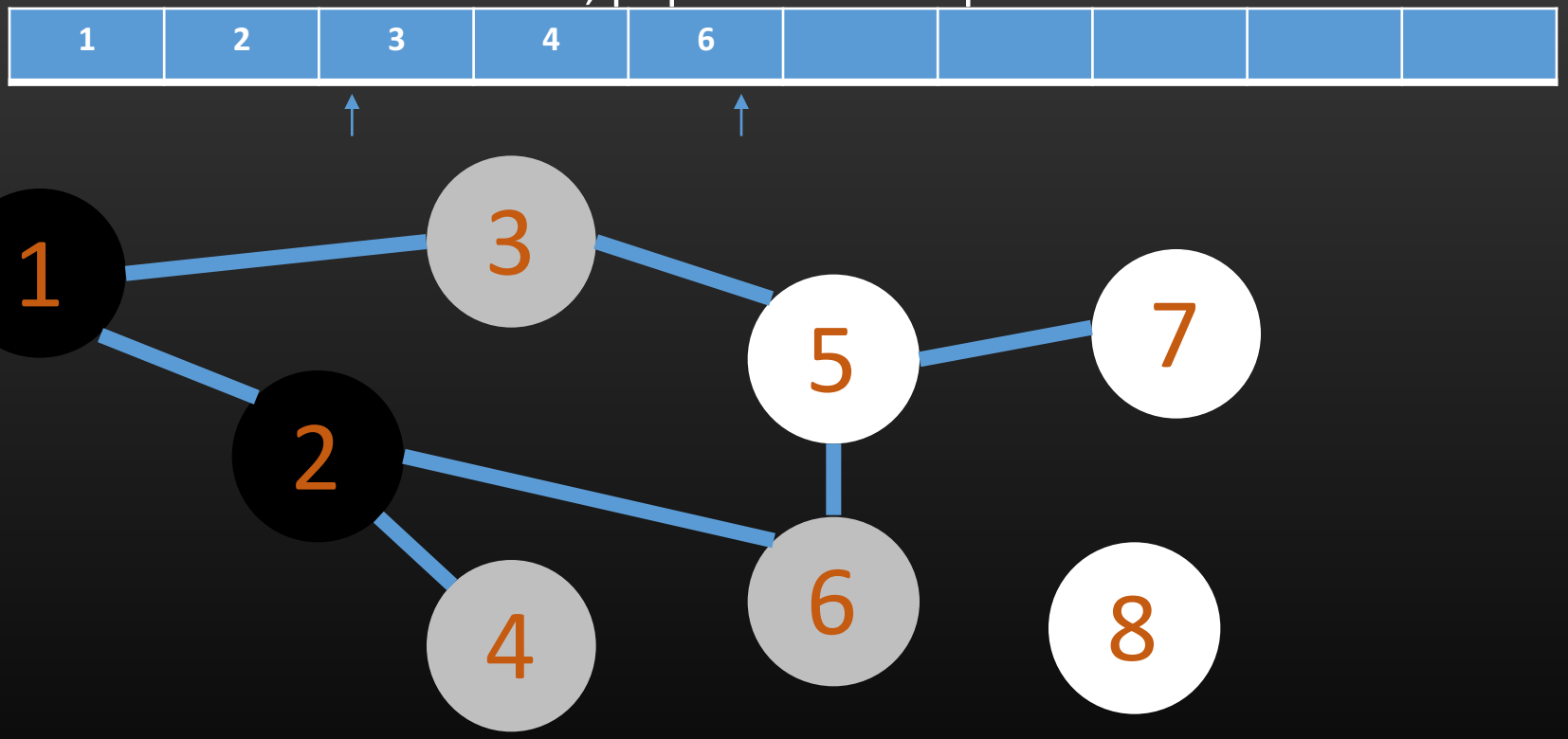
Breadth-first Search - example

- Mark node 4 and 6 as visited and push them into the queue



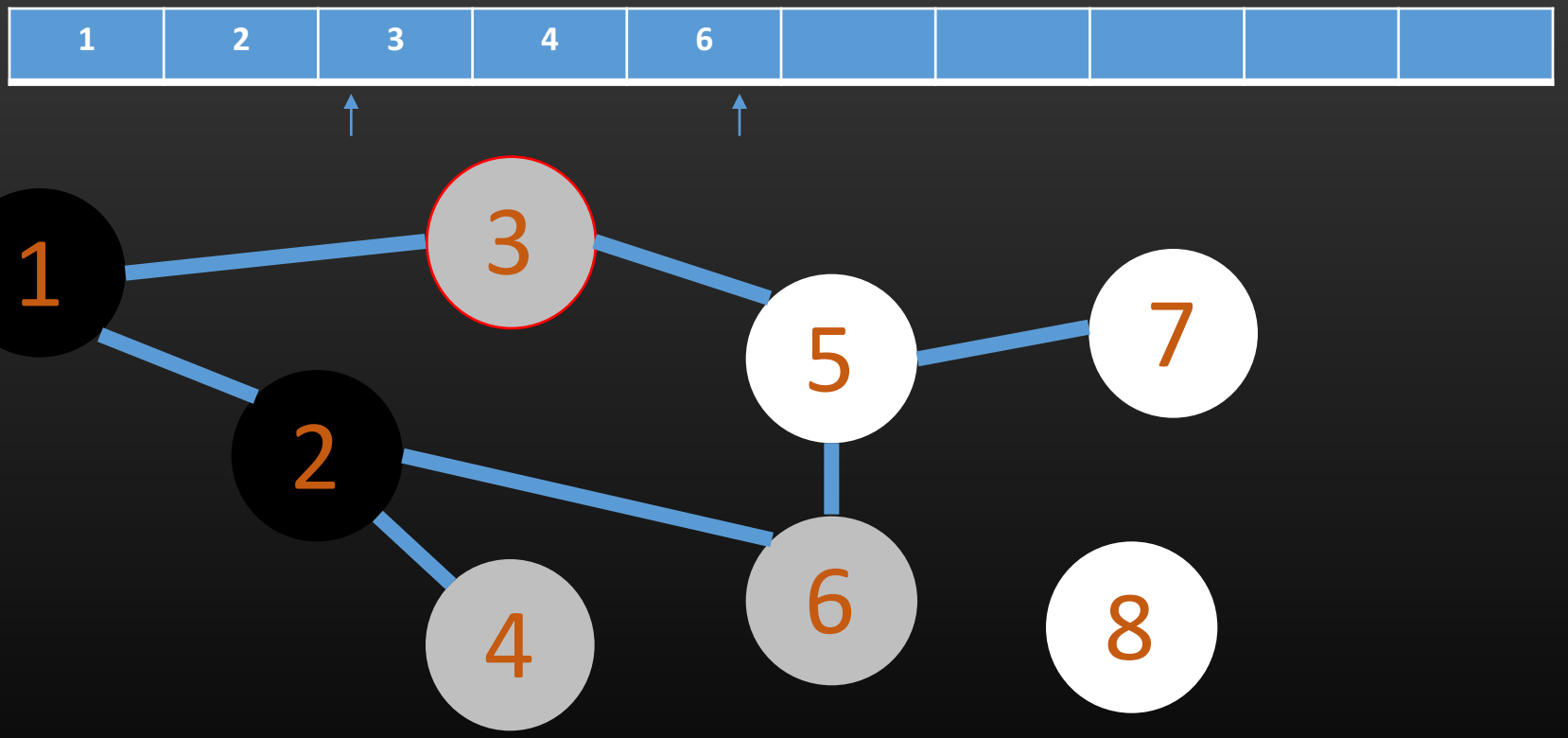
Breadth-first Search - example

- BFS on node 2 is finished, pop 2 from the queue



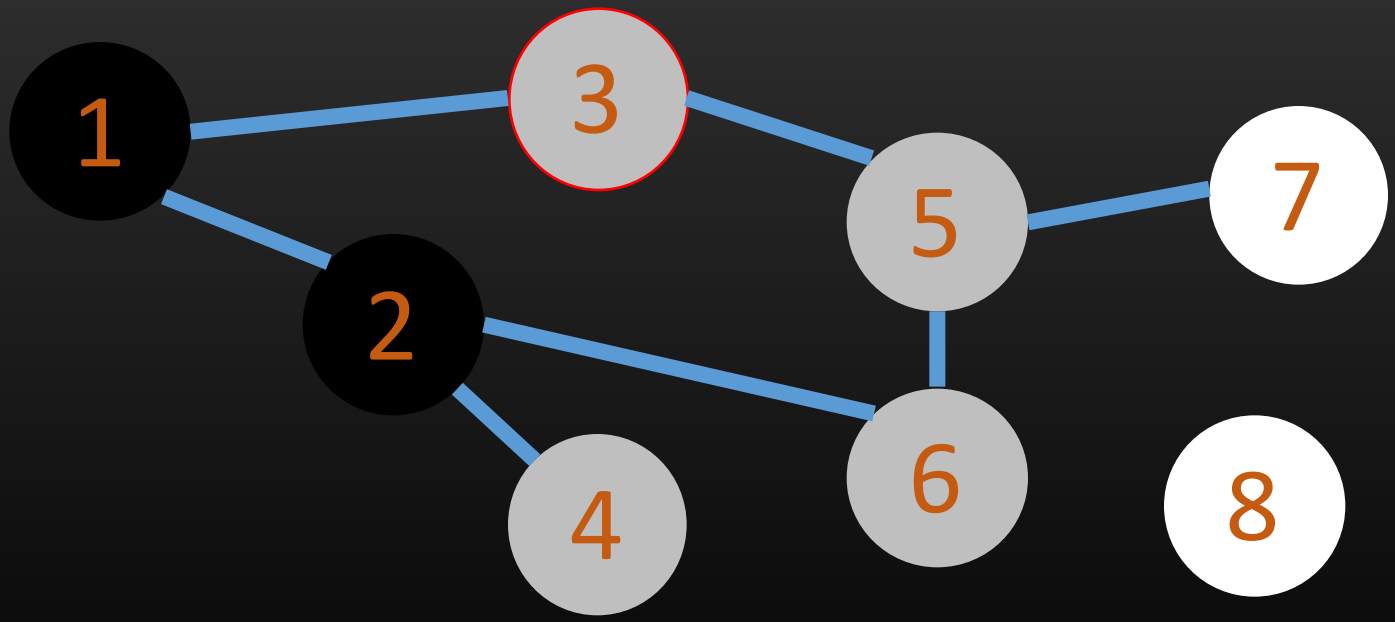
Breadth-first Search - example

- … and so on



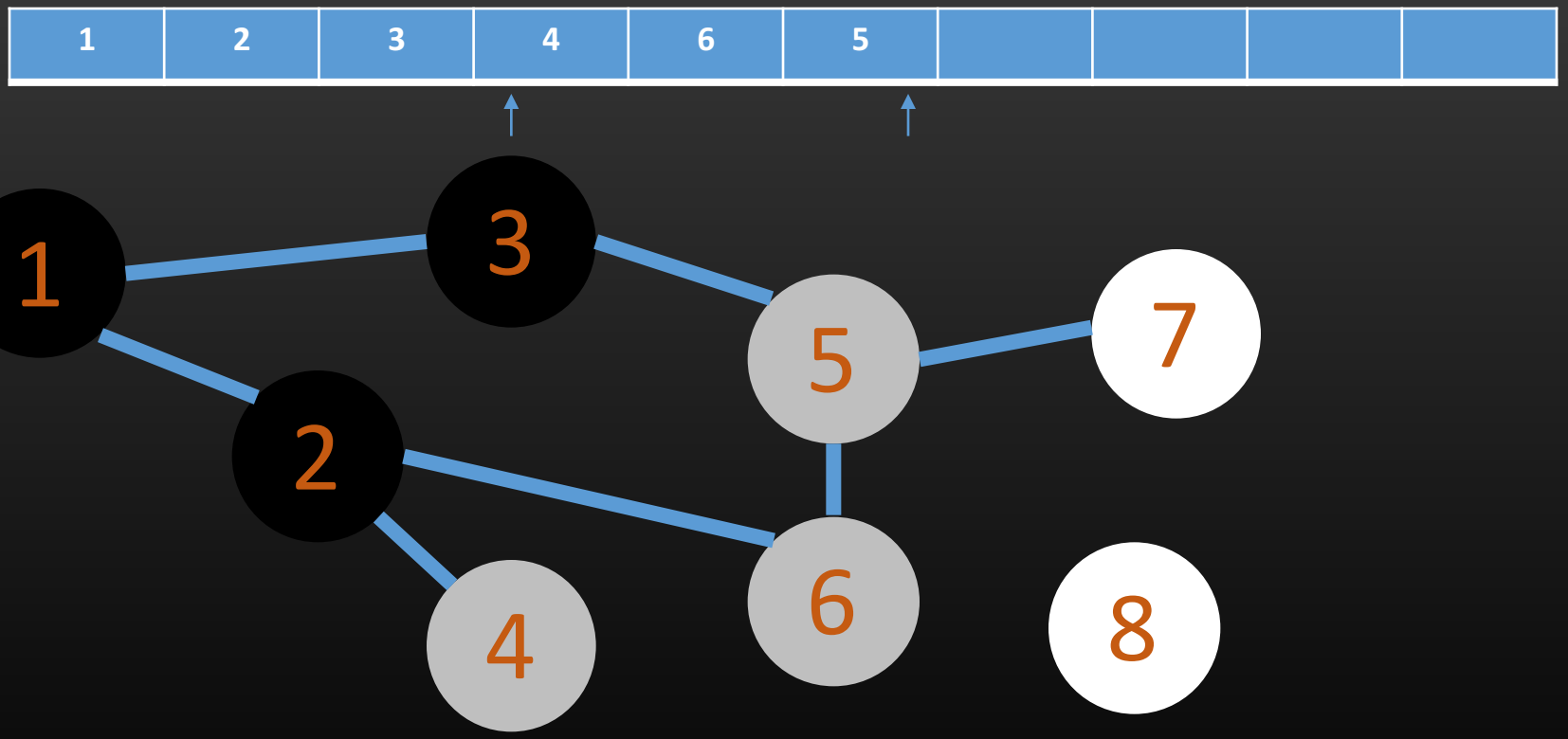
Breadth-first Search - example

- … and so on



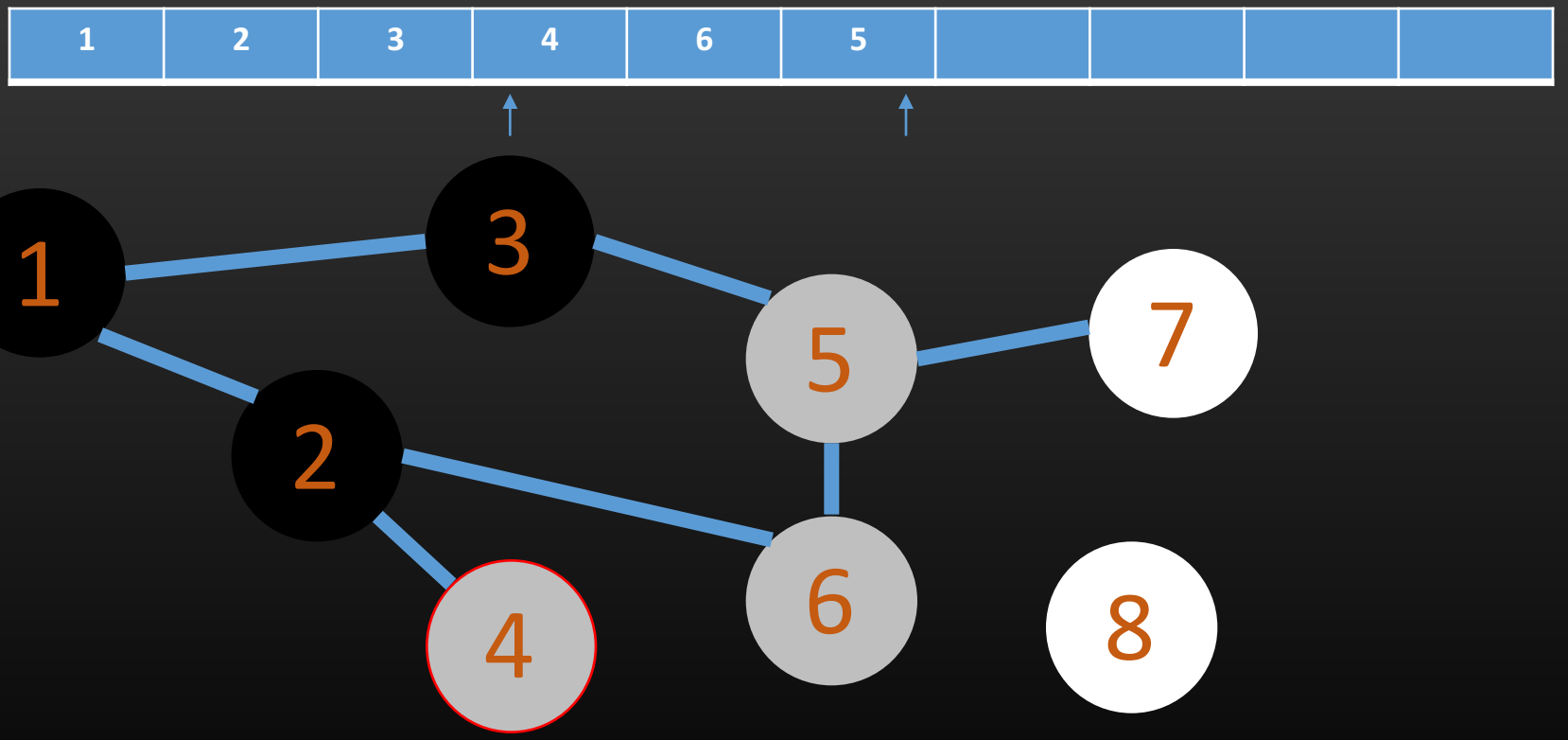
Breadth-first Search - example

- … and so on



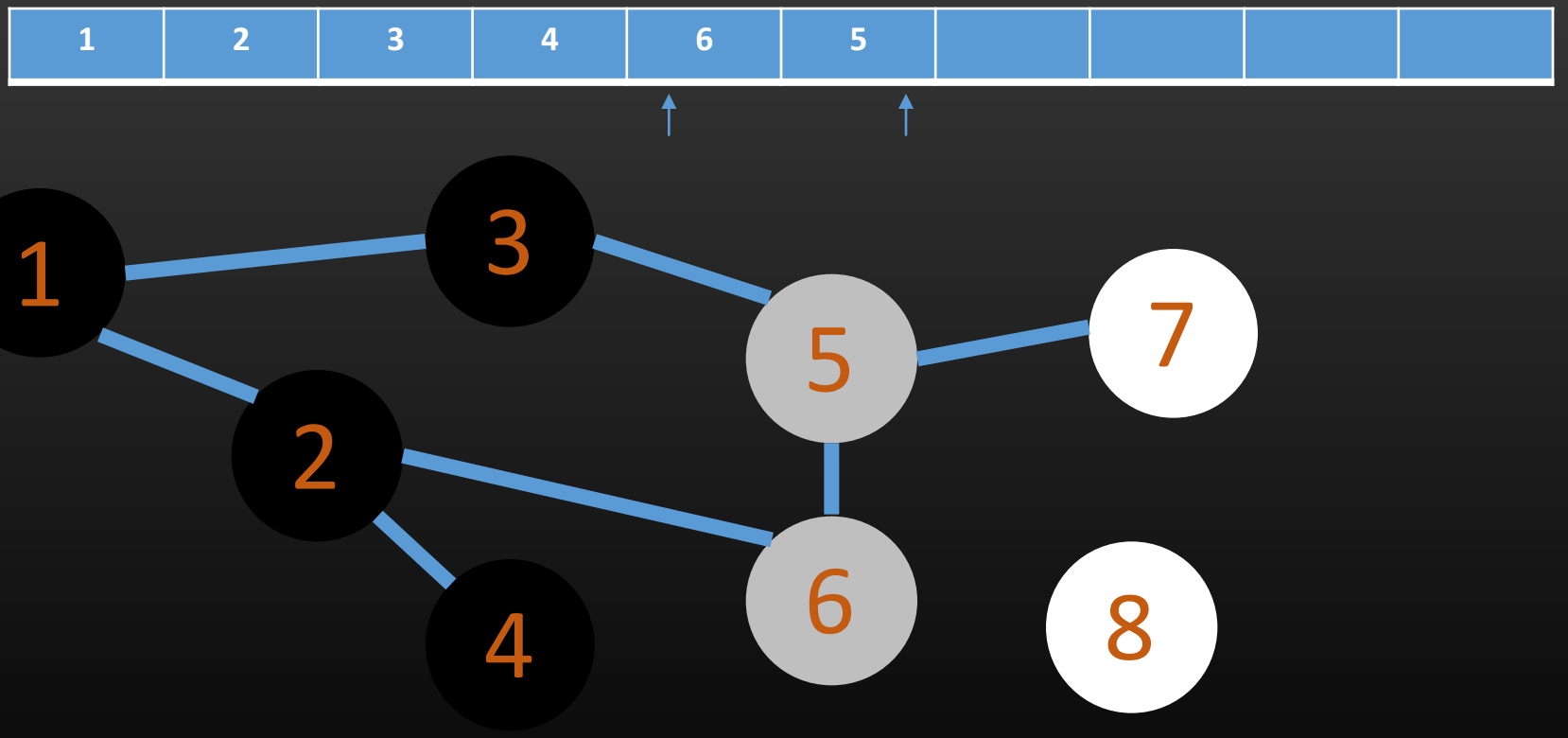
Breadth-first Search - example

- … and so on



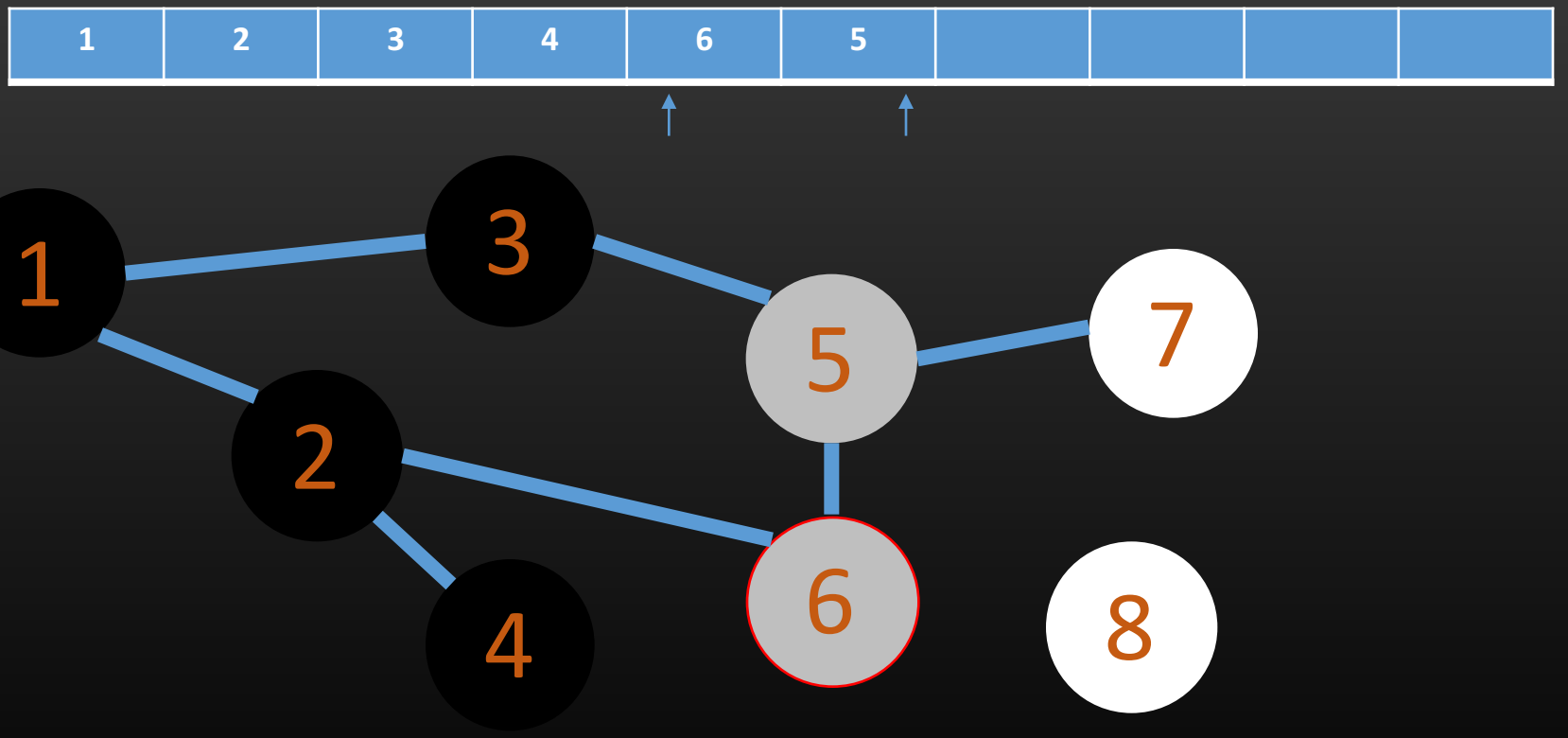
Breadth-first Search - example

- … and so on



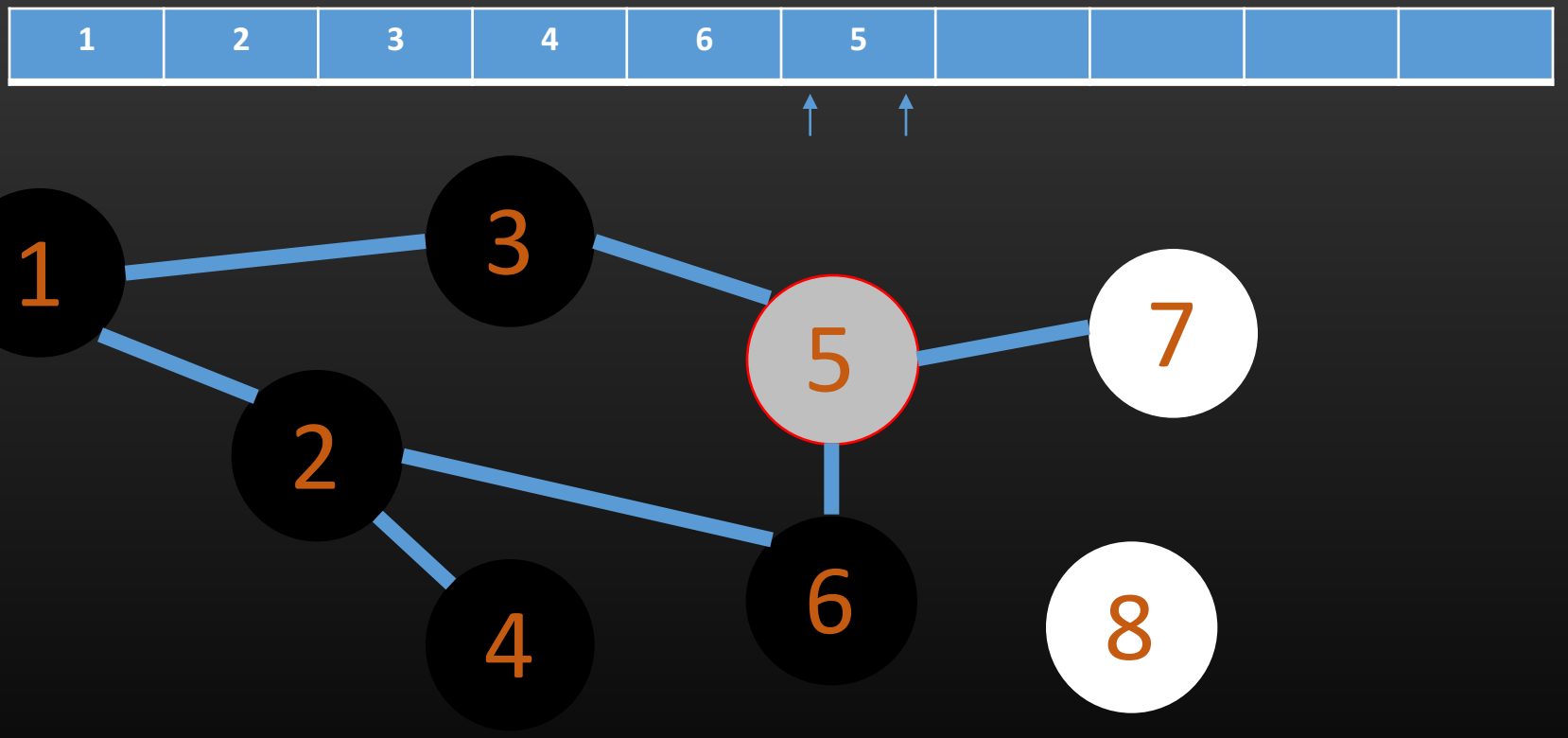
Breadth-first Search - example

- … and so on



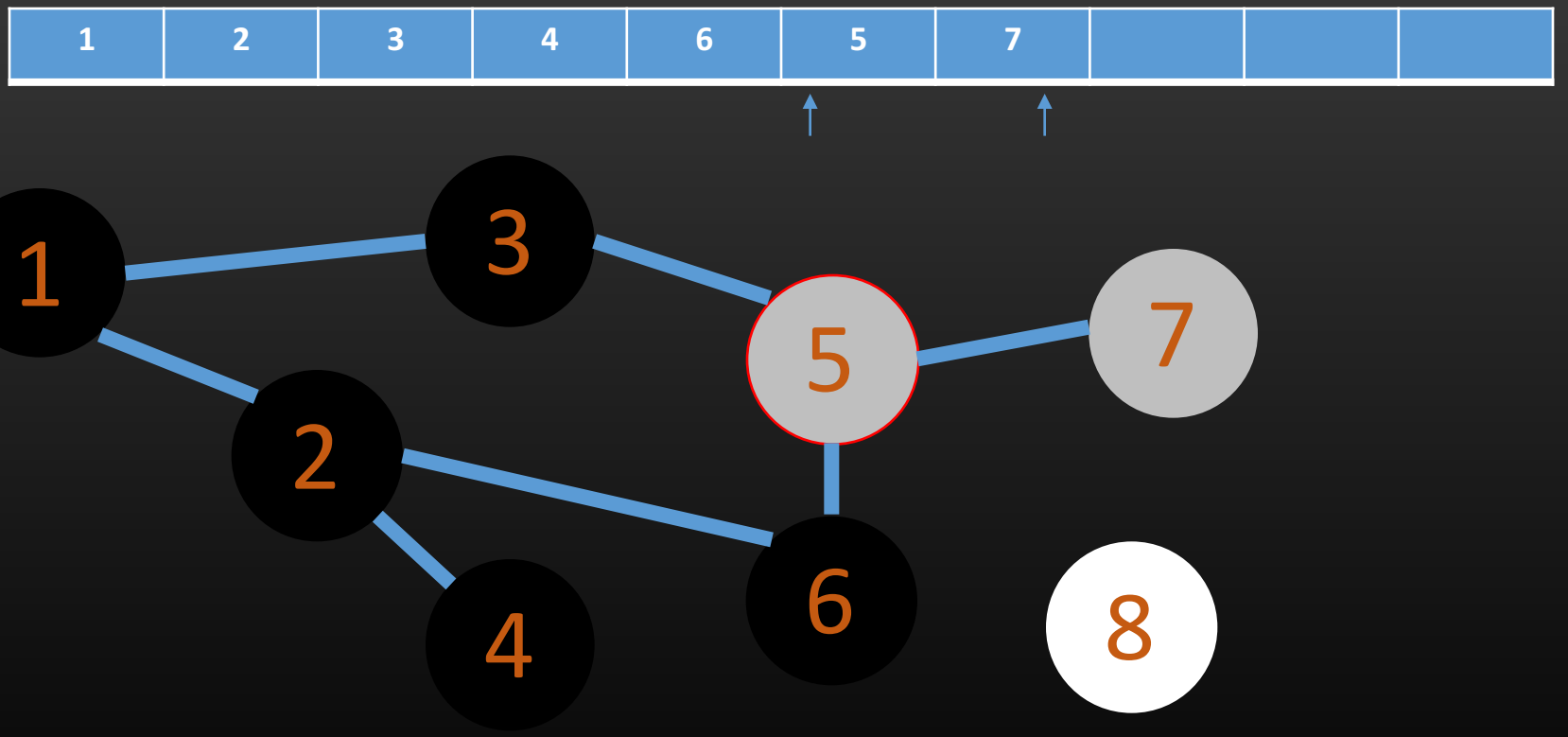
Breadth-first Search - example

- … and so on



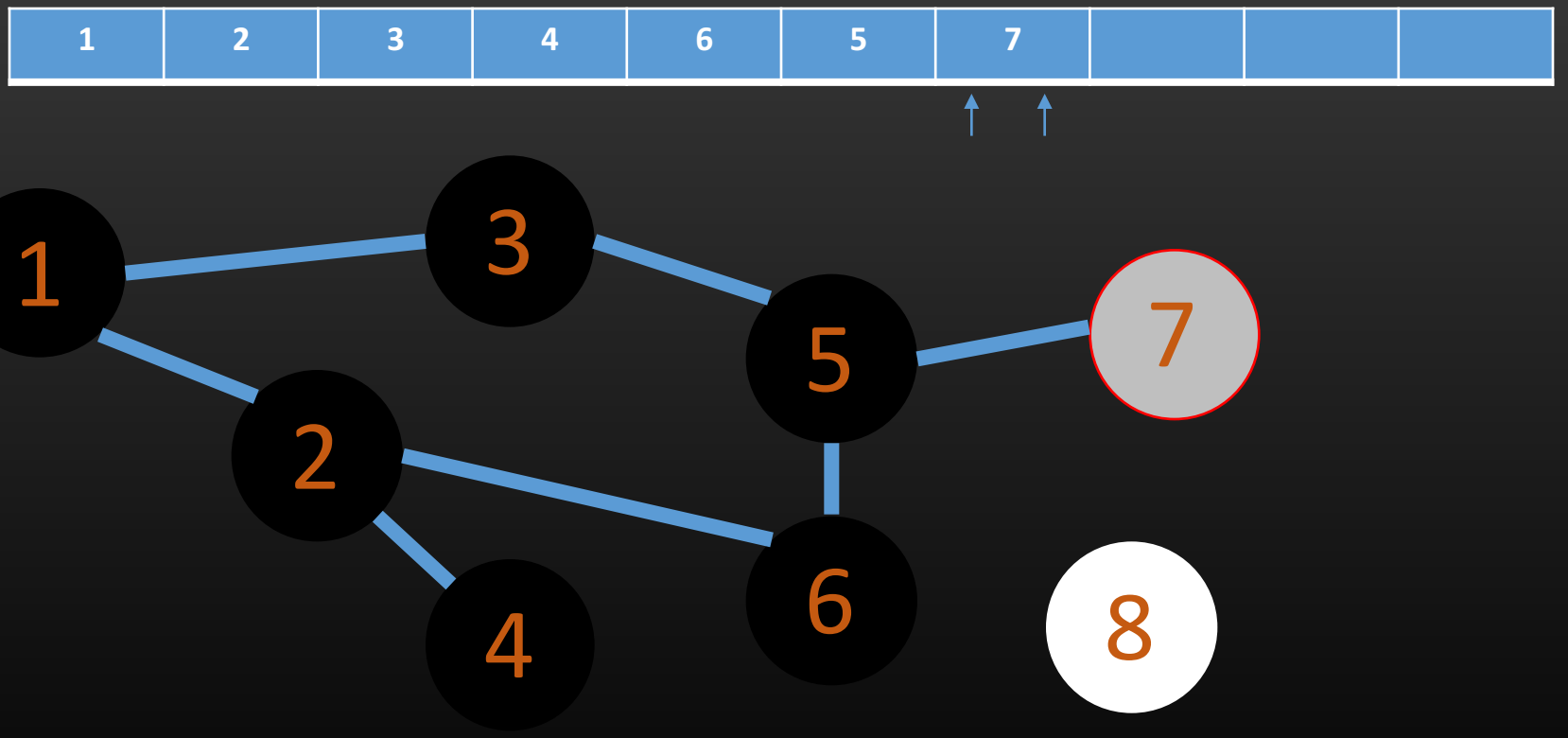
Breadth-first Search - example

- … and so on



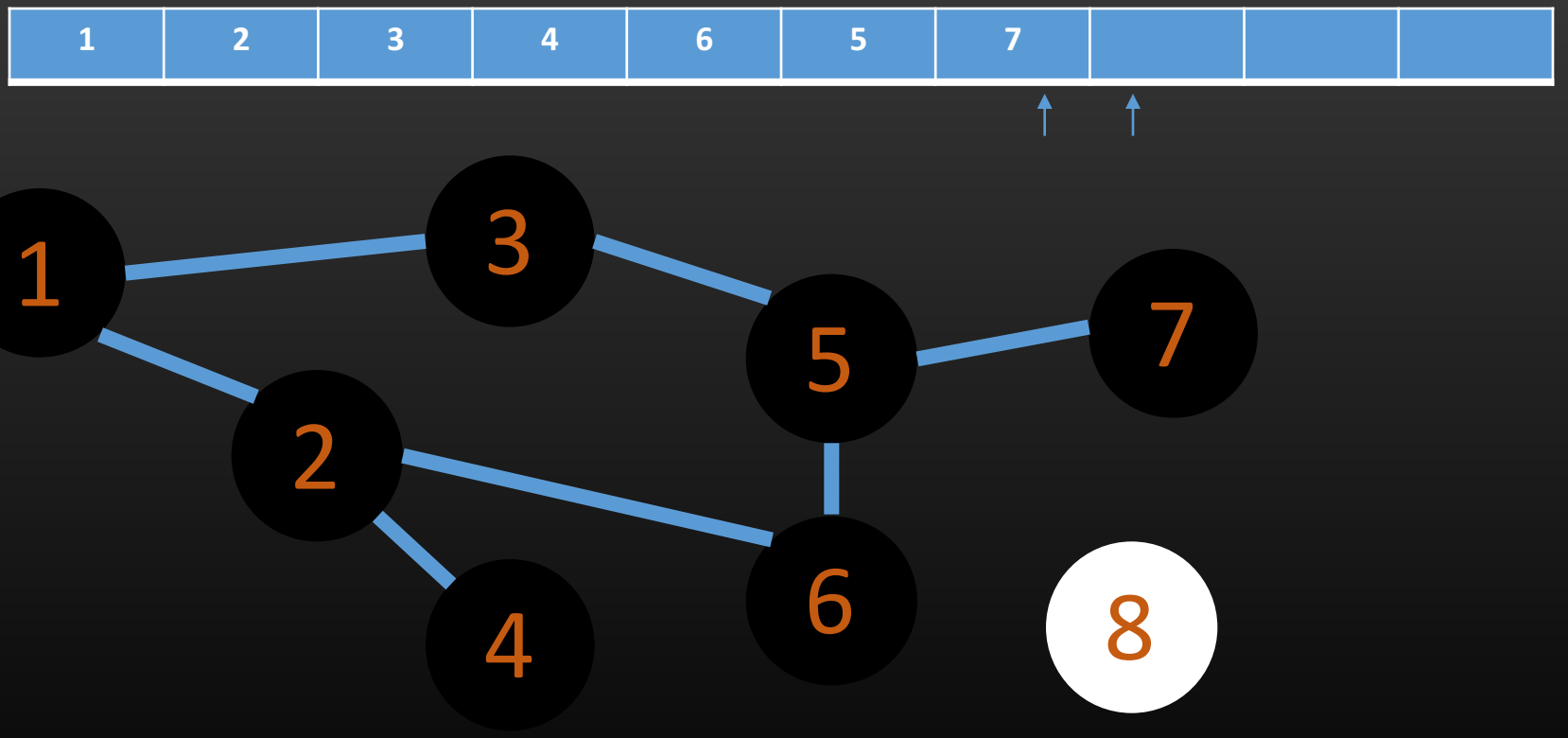
Breadth-first Search - example

- … and so on



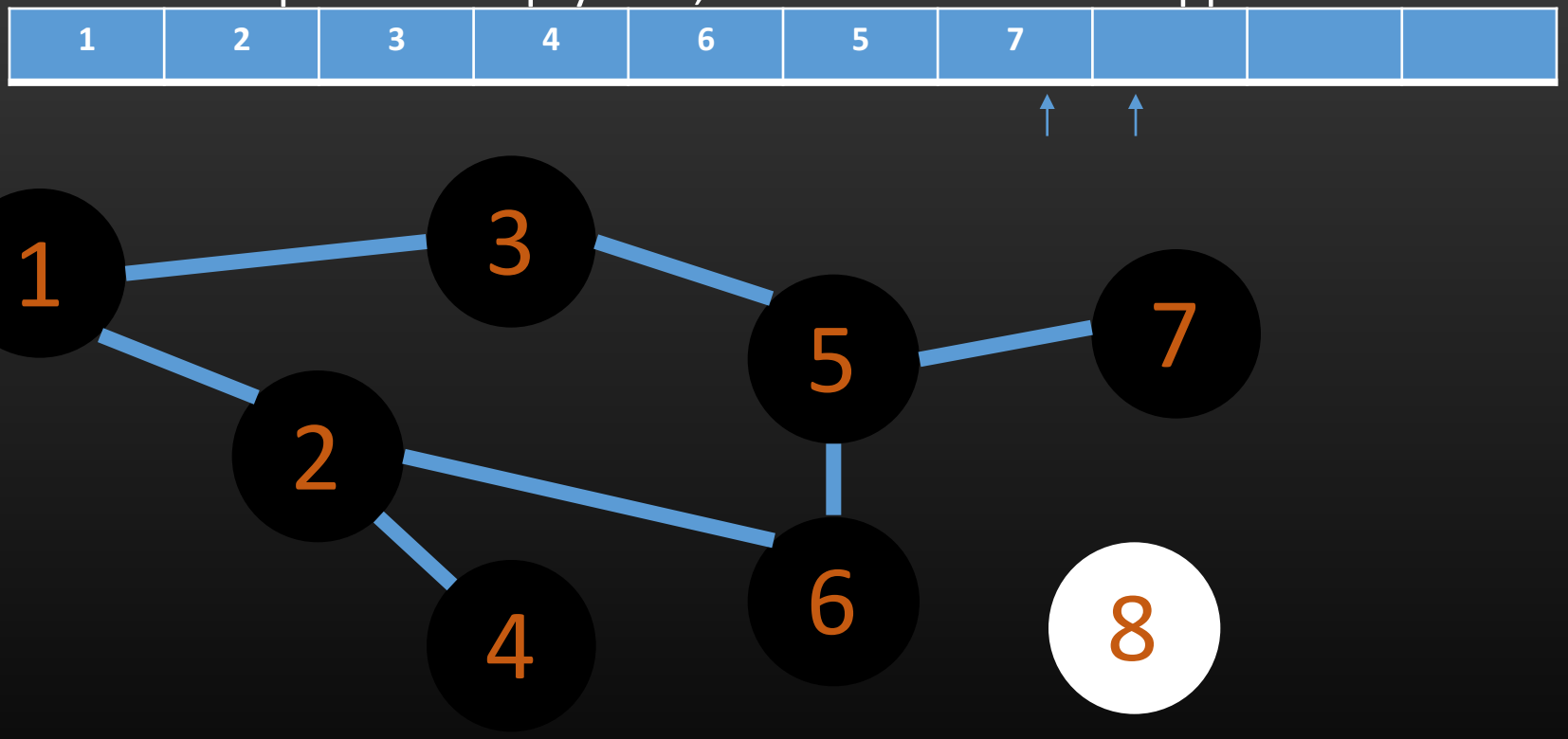
Breadth-first Search - example

- … and so on



Breadth-first Search - example

- Since the queue is empty now, the search can be stopped



Breadth-first Search - code

```
Procedure bfs(vertex v){
    mark v is visited
    push v into the queue
    while the queue is not empty do
        t = front of the queue
        for all vertex w adjacent to t do
            if w is not visited
                mark w as visited
                push w into the queue
        pop t from the queue
}
```

Breadth-first Search

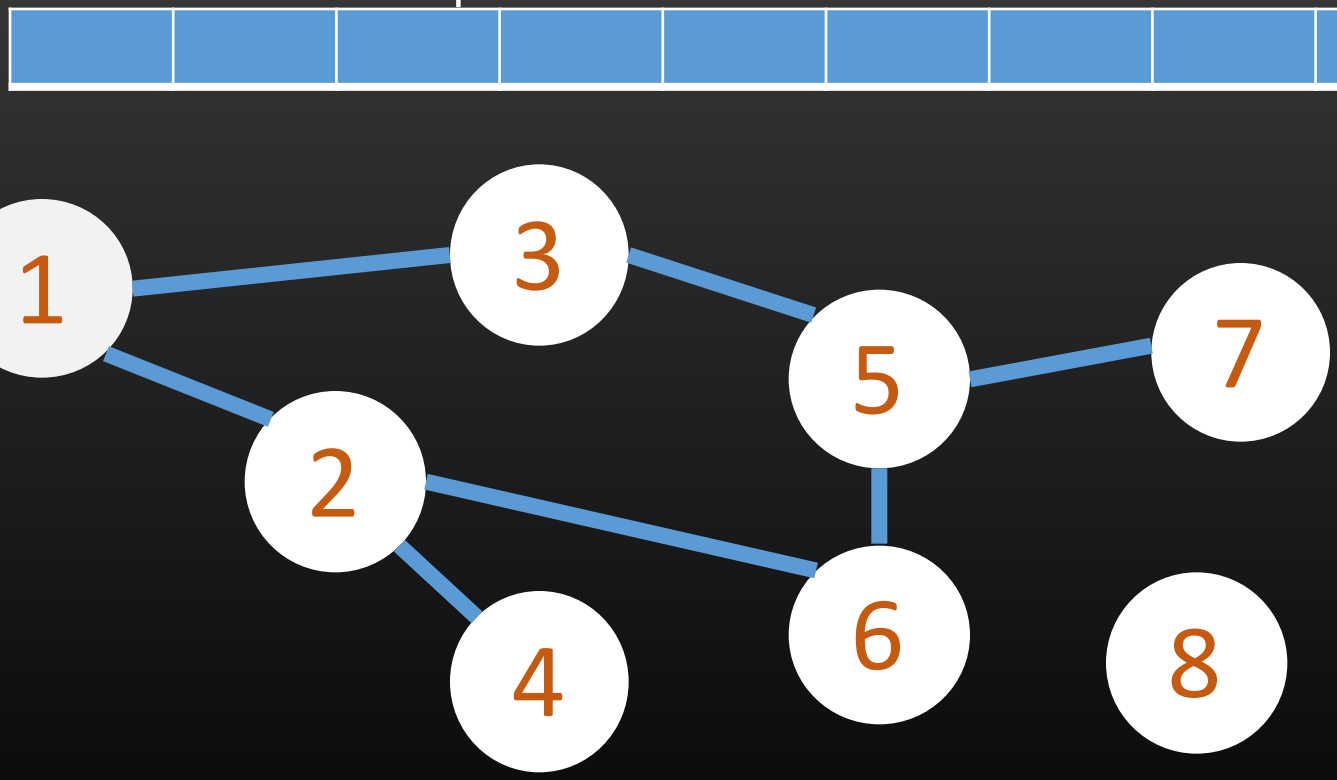
- Time Complexity:
  - $O(|V|^2)$ when using adjacency matrix
  - $O(|V|+|E|)$ when using adjacency list / edge list

Breadth-first Search - application

- Find the shortest path of all nodes from one node, with path length measured by number of edges

Breadth-first Search - example

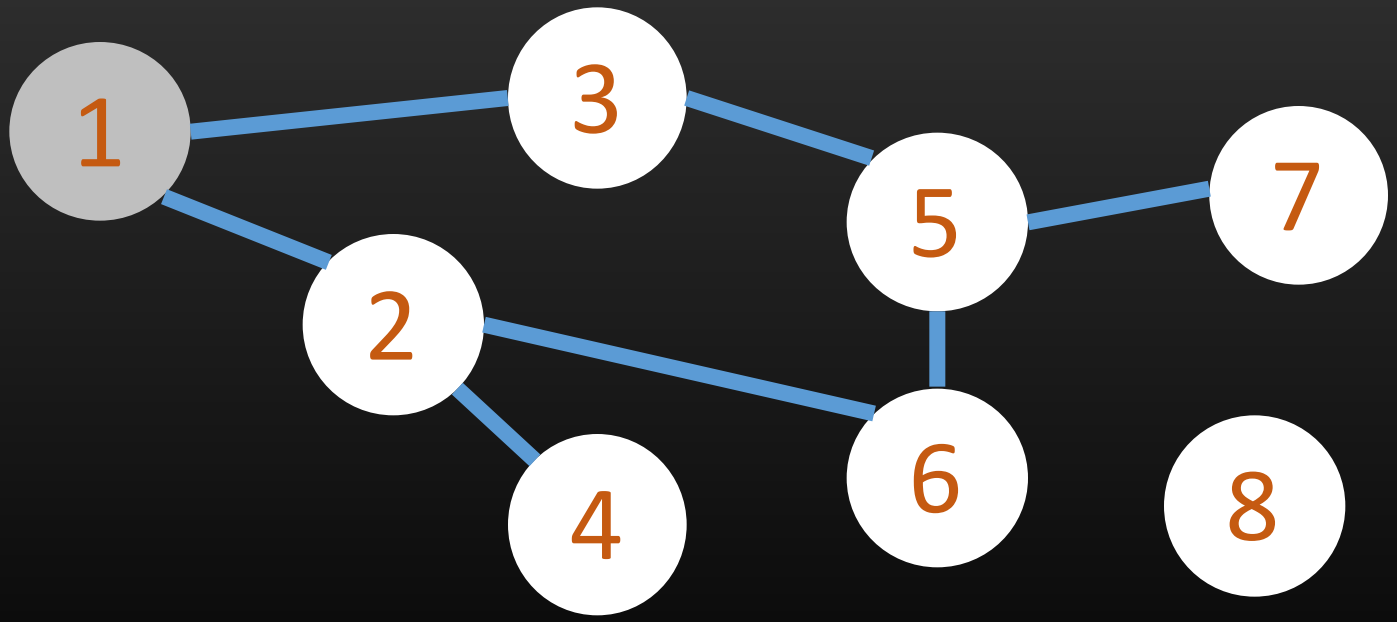
- Find the shortest paths from node 1



Breadth-first Search - example

1									

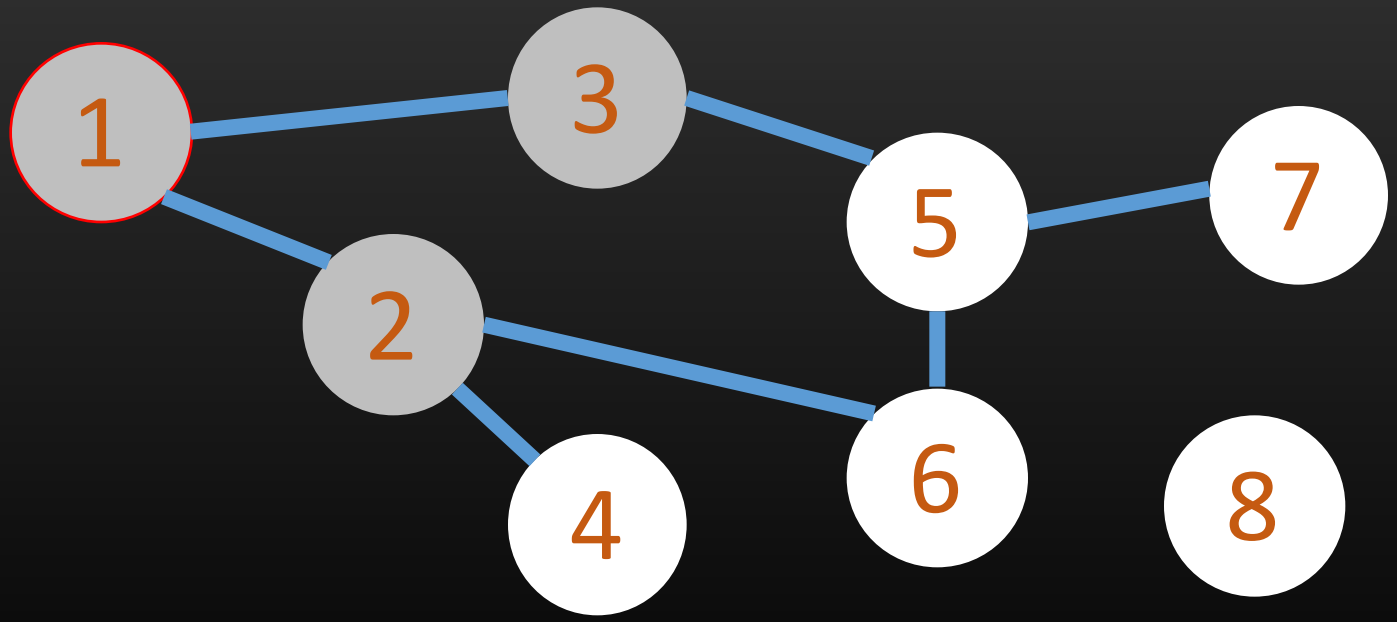
Node	1	2	3	4	5	6	7	8
Distance	0	∞	∞	∞	∞	∞	∞	∞



Breadth-first Search - example

1	2	3							

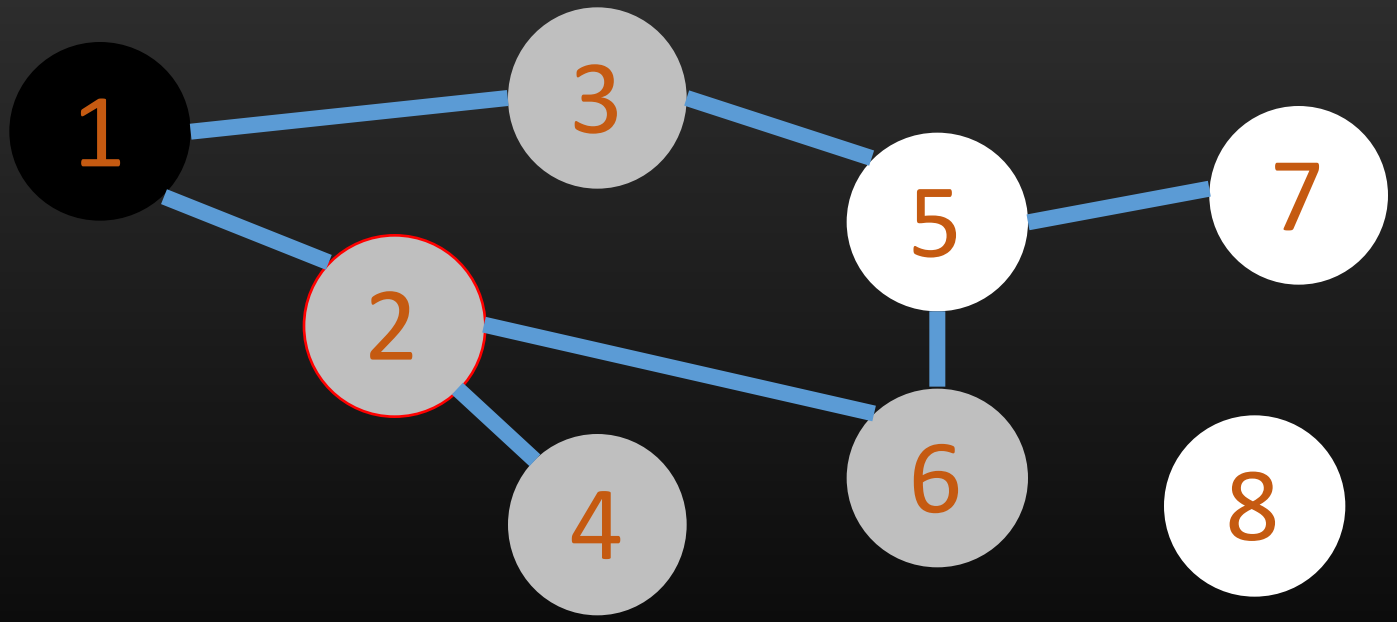
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	∞	∞	∞	∞	∞



Breadth-first Search - example

1	2	3	4	6					

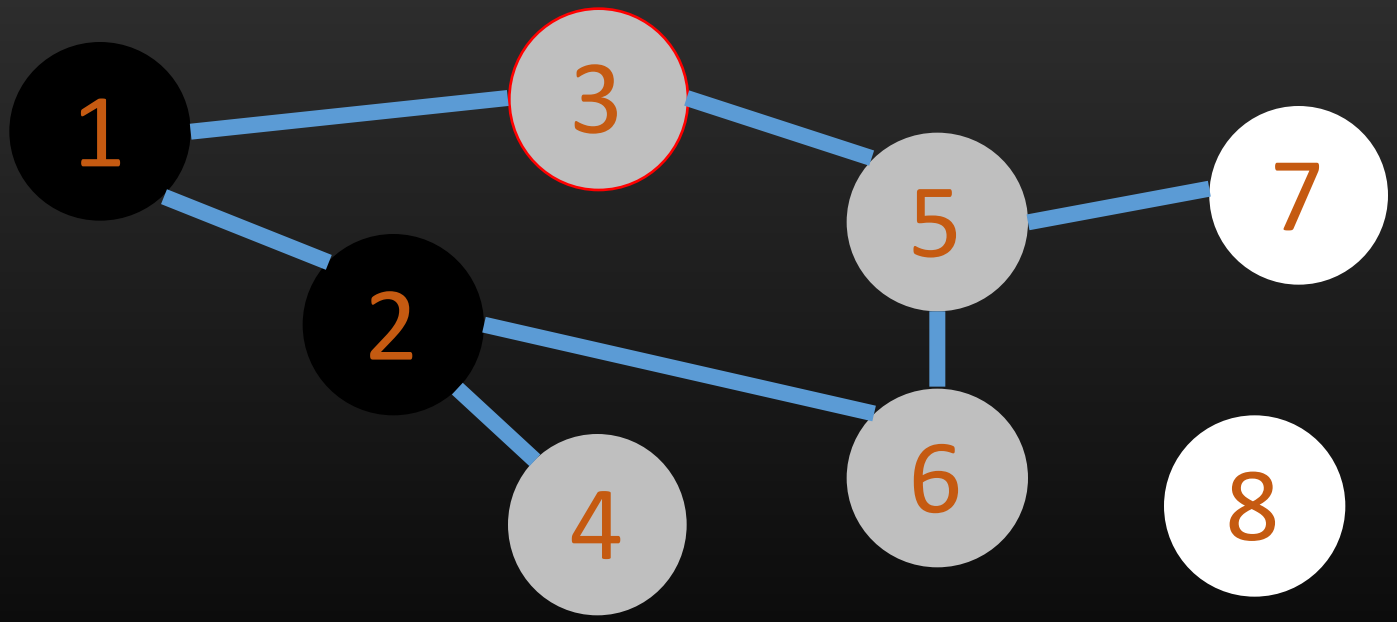
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	∞	2	∞	∞



Breadth-first Search - example

1	2	3	4	6	5				

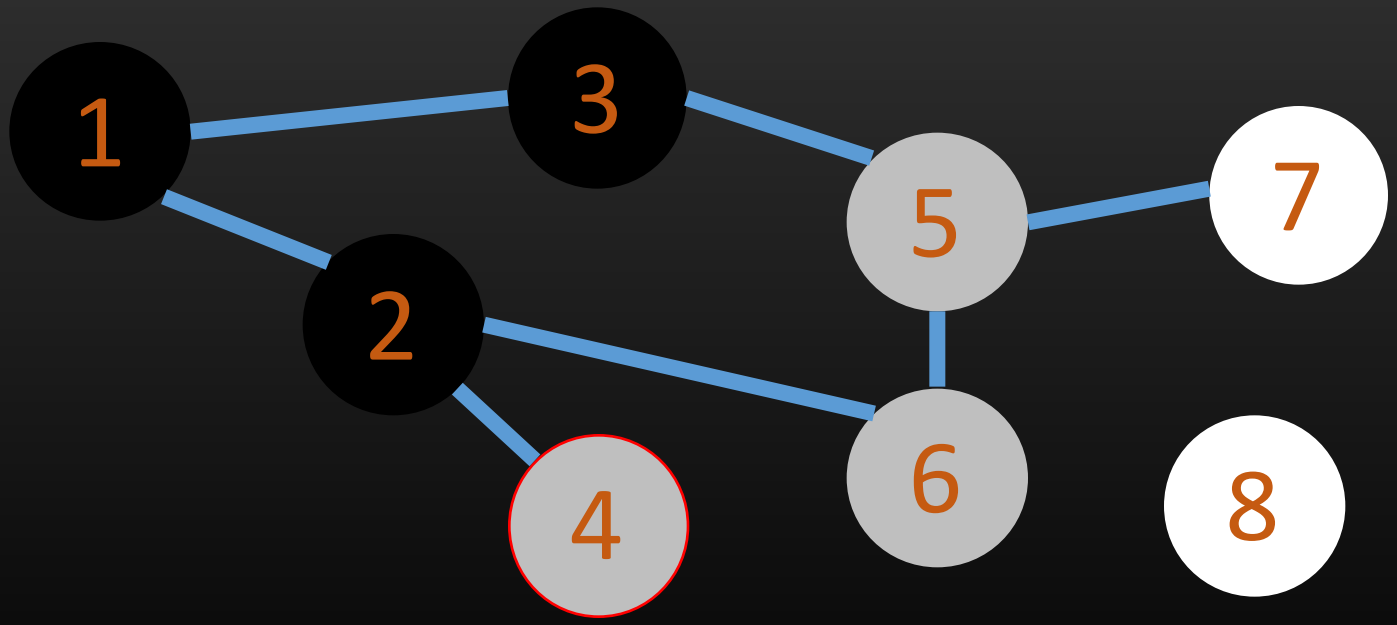
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	∞	∞



Breadth-first Search - example

1	2	3	4	6	5				

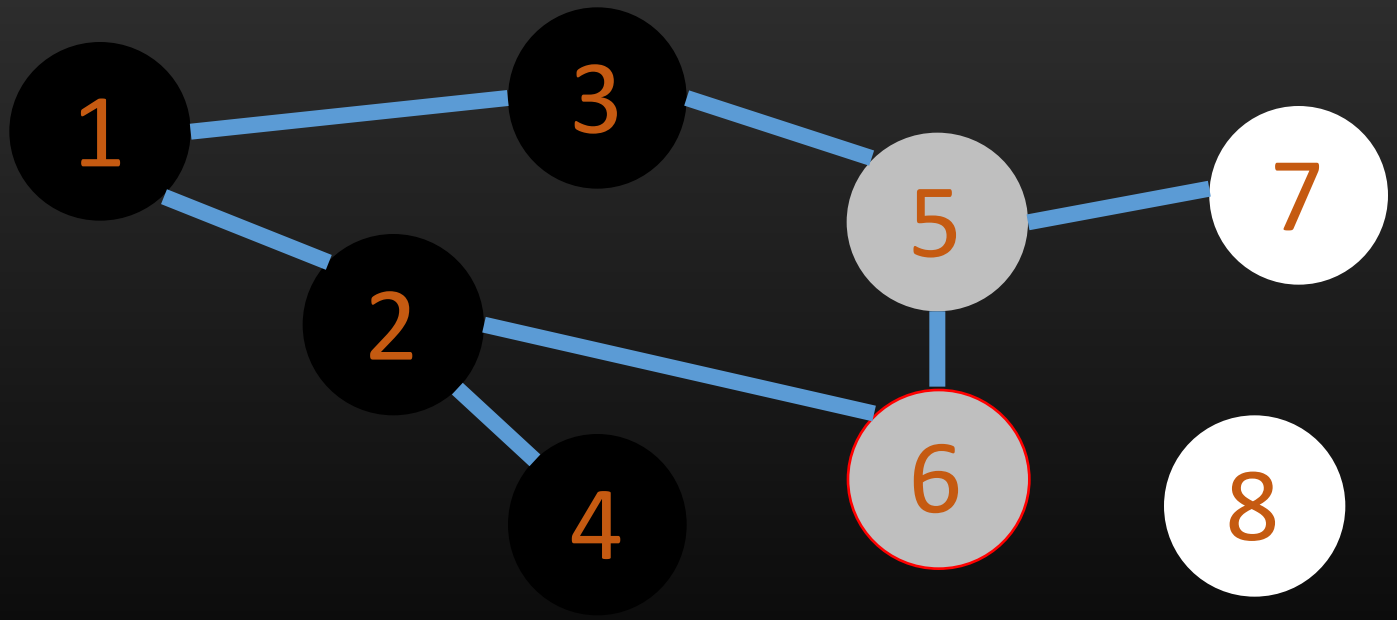
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	∞	∞



Breadth-first Search - example

1	2	3	4	6	5				

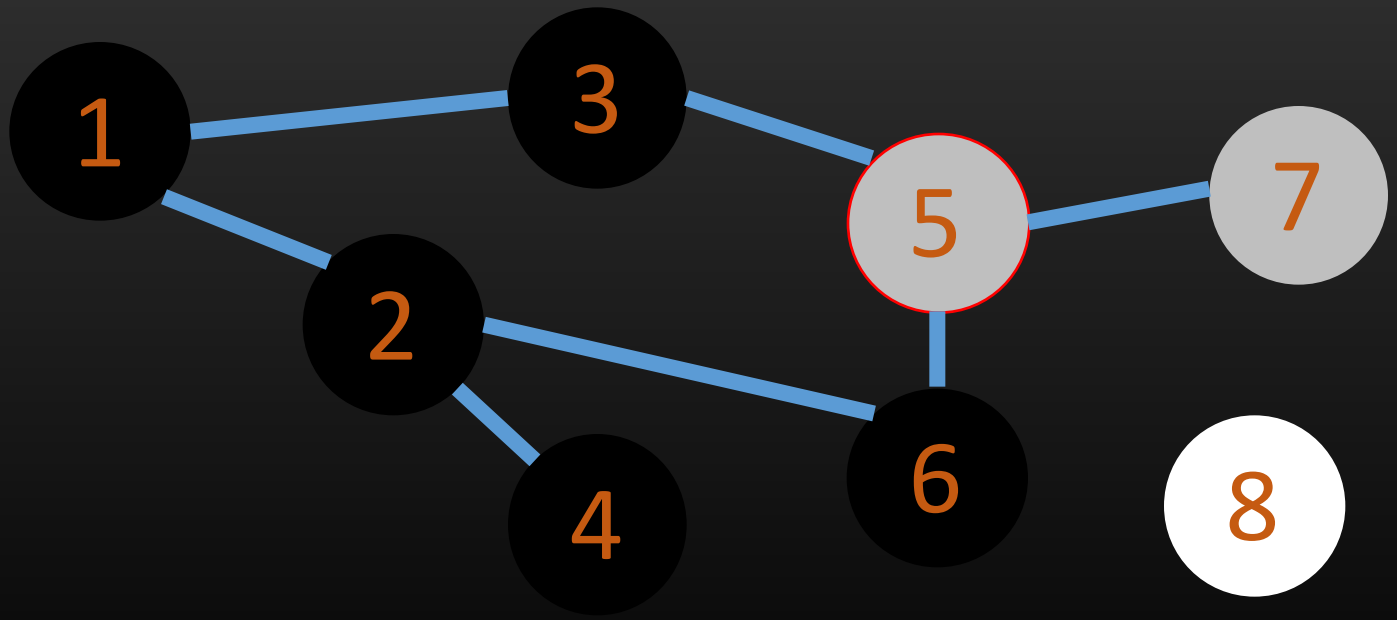
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	∞	∞



Breadth-first Search - example

1	2	3	4	6	5	7			

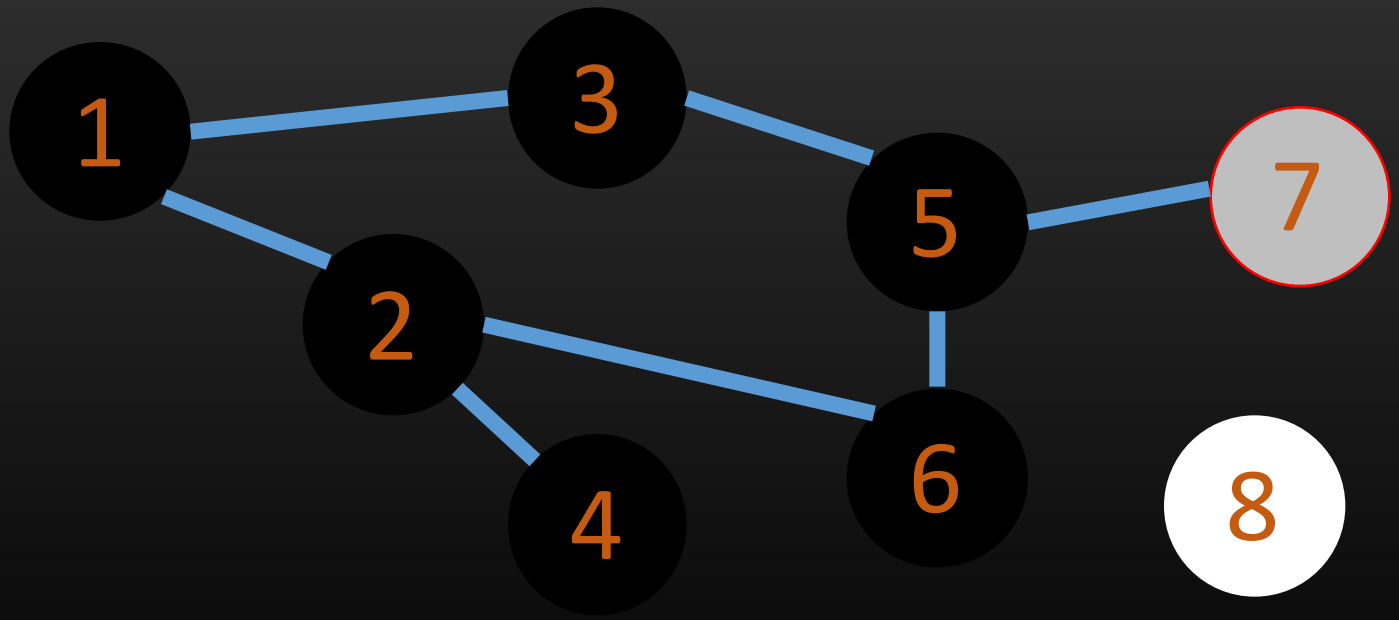
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	3	∞



Breadth-first Search - example

1	2	3	4	6	5	7			

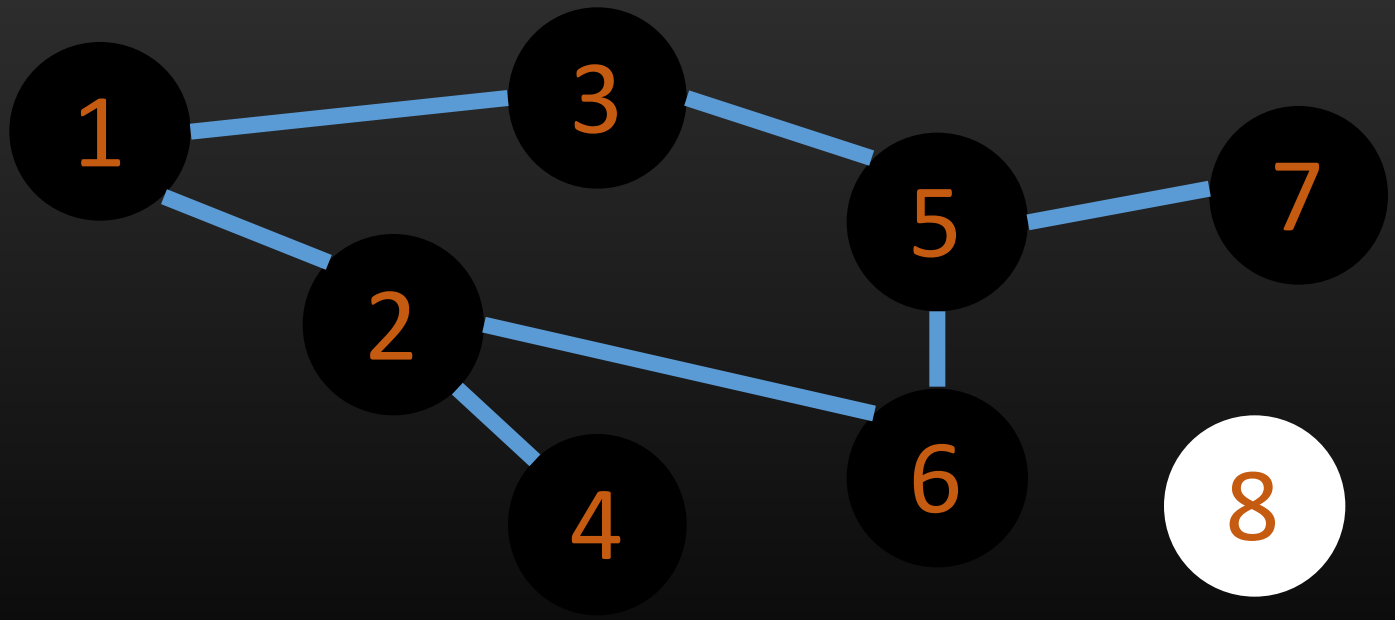
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	3	∞



Breadth-first Search - example

1	2	3	4	6	5	7			

Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	3	∞



Multisource BFS

- Sometimes you may want to start searching from more than one source

- Method 1:
- Perform BFS once for each source

- Time Complexity: O(N*(|V|+|E|)), where N is the number of sources
- Too slow if there are many sources

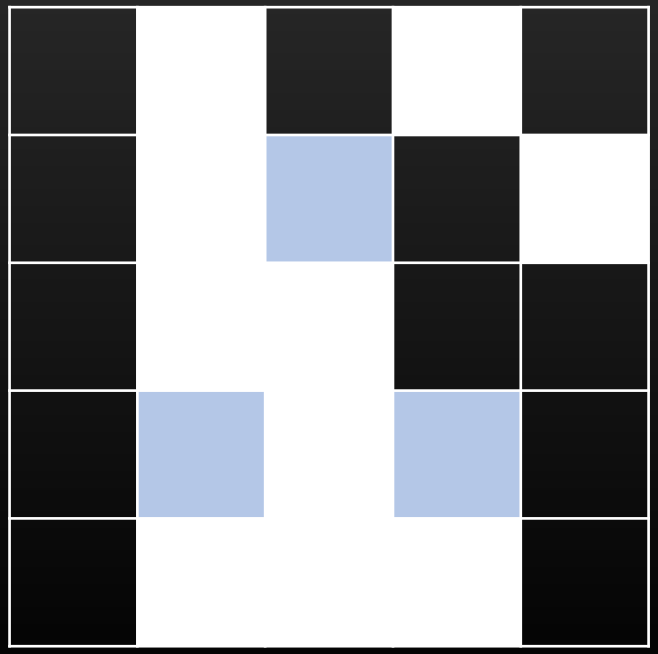
Multisource BFS

- Method 2:
- Push all sources into the queue first
- Perform BFS once
- Time Complexity: O(|V|+|E|)

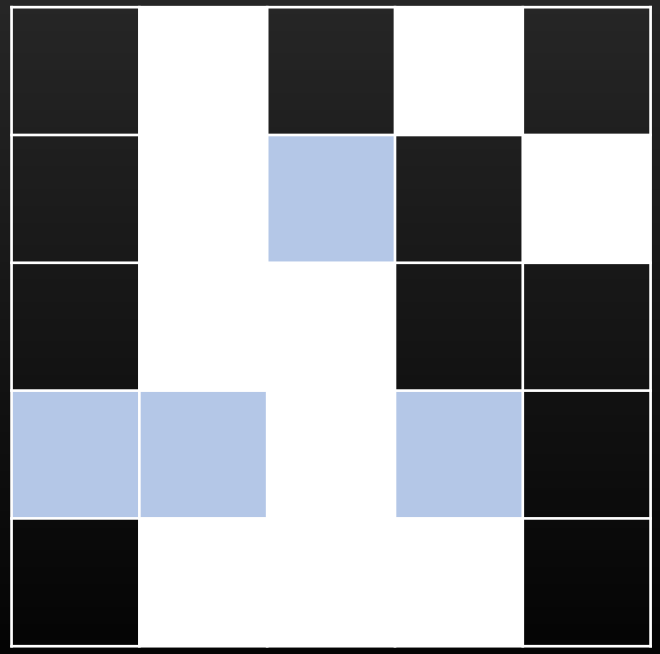
Multisource BFS

- Find the black area connected to (4, 2), (4, 4) and (2, 3)
- Push (4, 2), (4, 4) and (2, 3) into the queue and marked them as visited

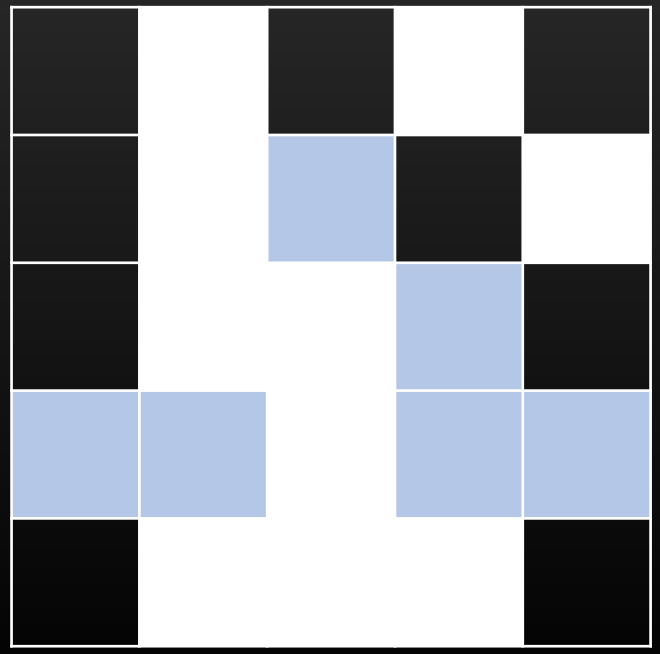
4, 2	4, 4	2, 3											



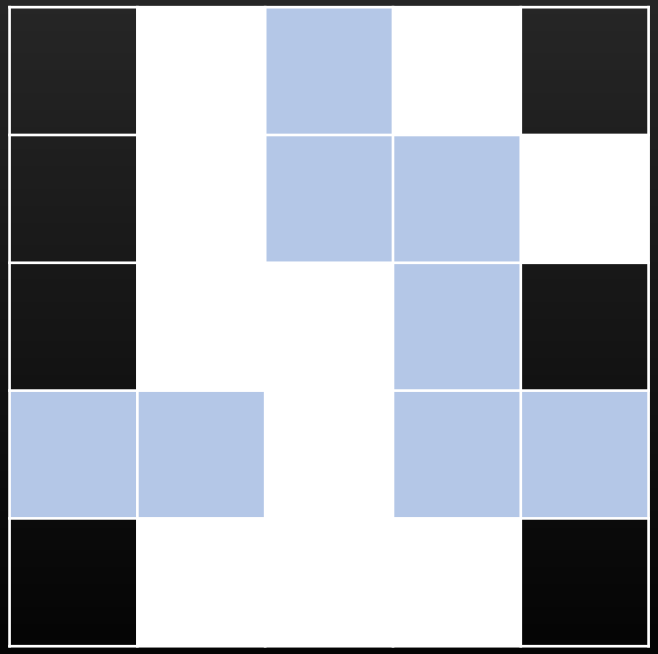
Multisource BFS



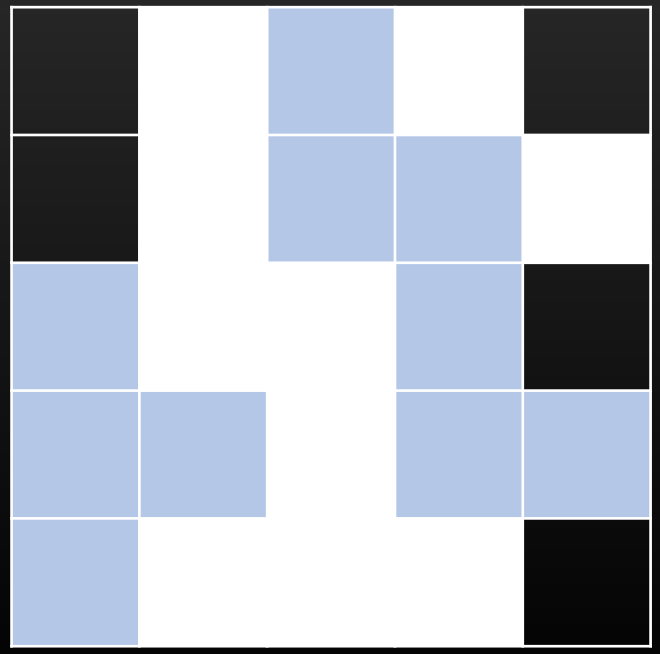
Multisource BFS



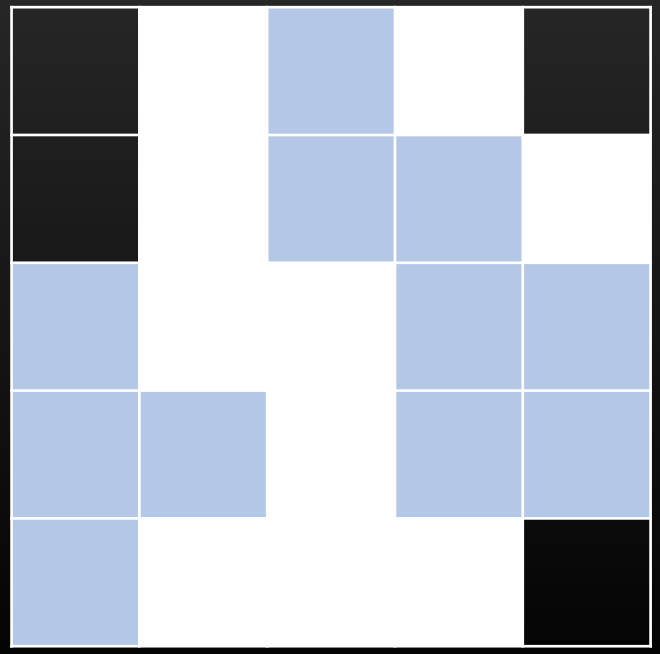
Multisource BFS



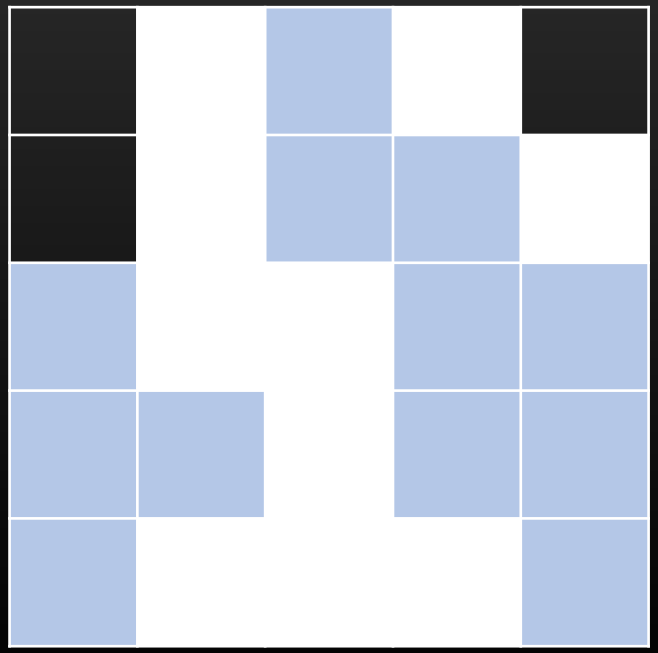
Multisource BFS



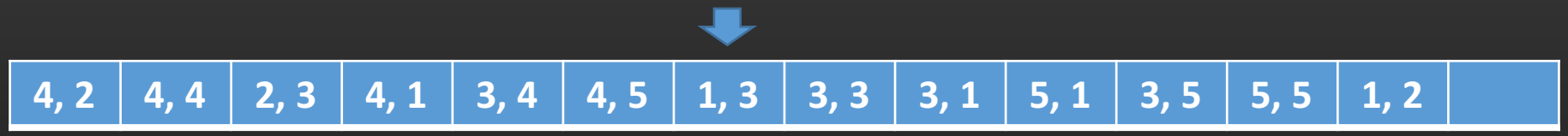
Multisource BFS

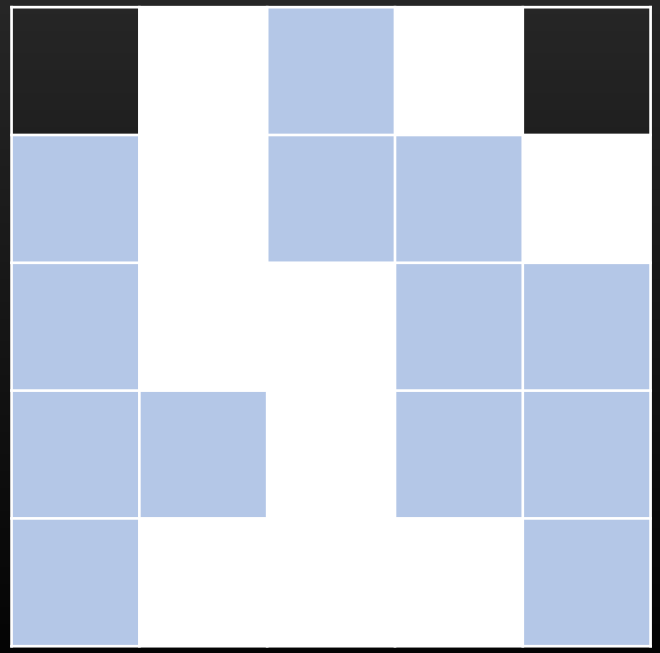


Multisource BFS



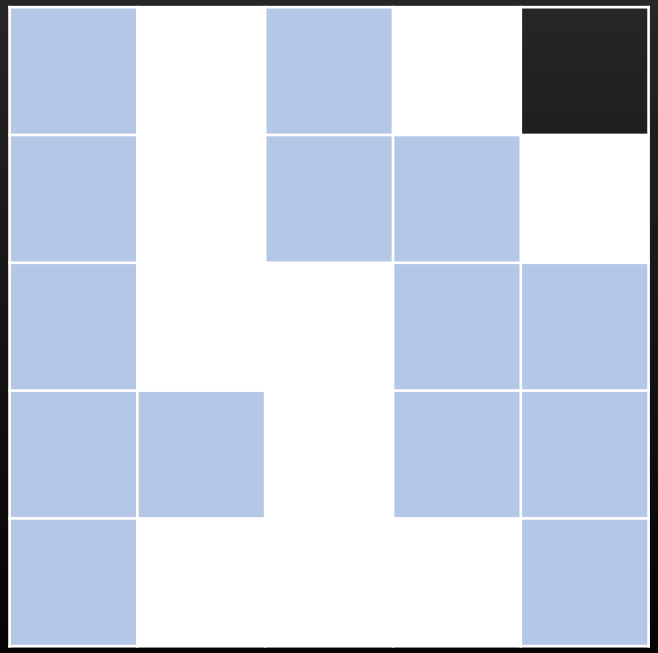
Multisource BFS





Multisource BFS

4, 2	4, 4	2, 3	4, 1	3, 4	4, 5	1, 3	3, 3	3, 1	5, 1	3, 5	5, 5	1, 2	1, 1



States

- A vertex can represent a state
- The edges are the possible transitions and the weights are the cost of the transitions

States – Example 1

- Water Jug Problem

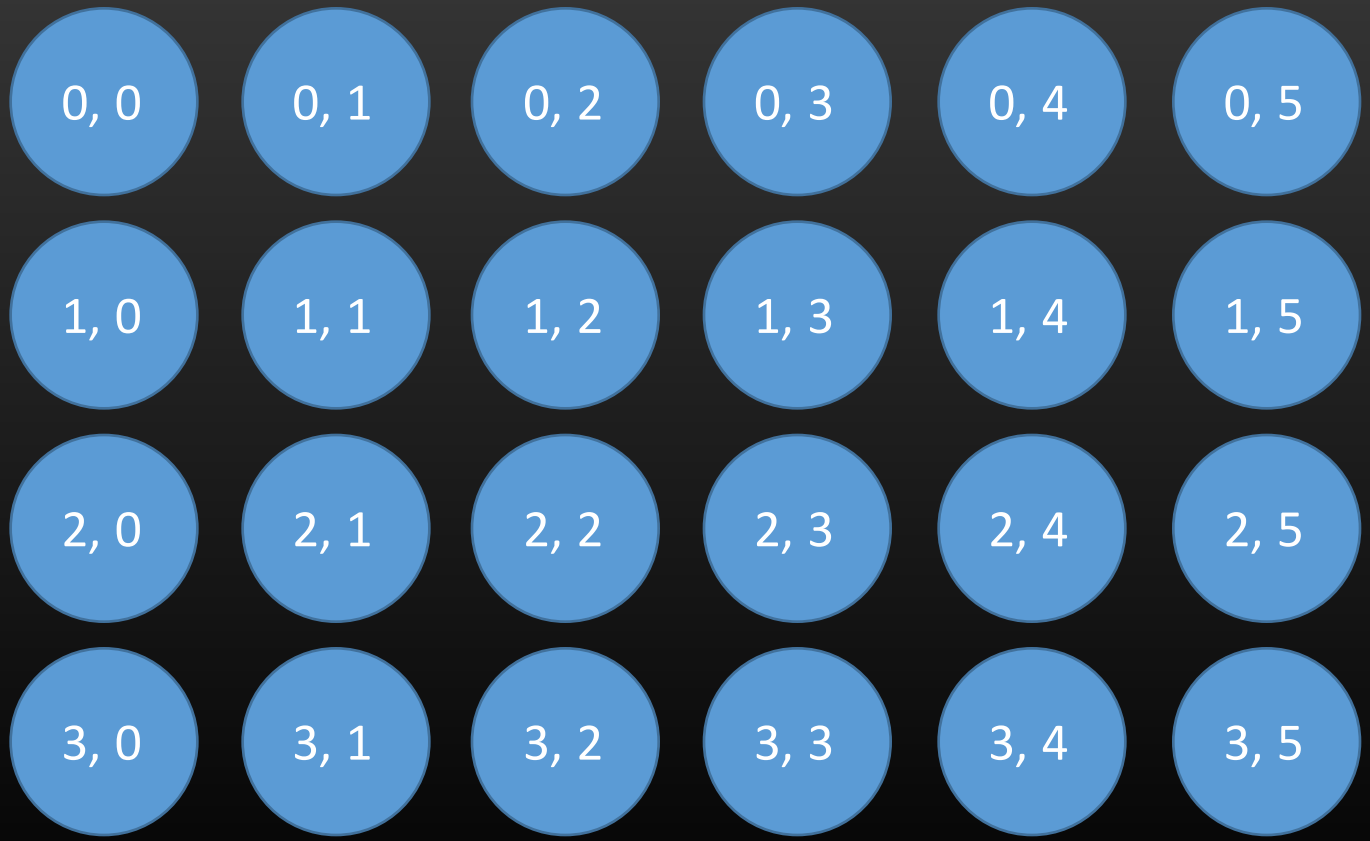
- There are two jugs with a capacity of N and M liters
- Both are empty initially
- Each time you can only do one of the three operations:
  - Empty a jug
  - Fill a jug completely
  - Pour water from one jug to another until either one jug becomes empty or the other becomes full
- Find the minimum number of operations to get a specific volume in one jug

States – Example 1

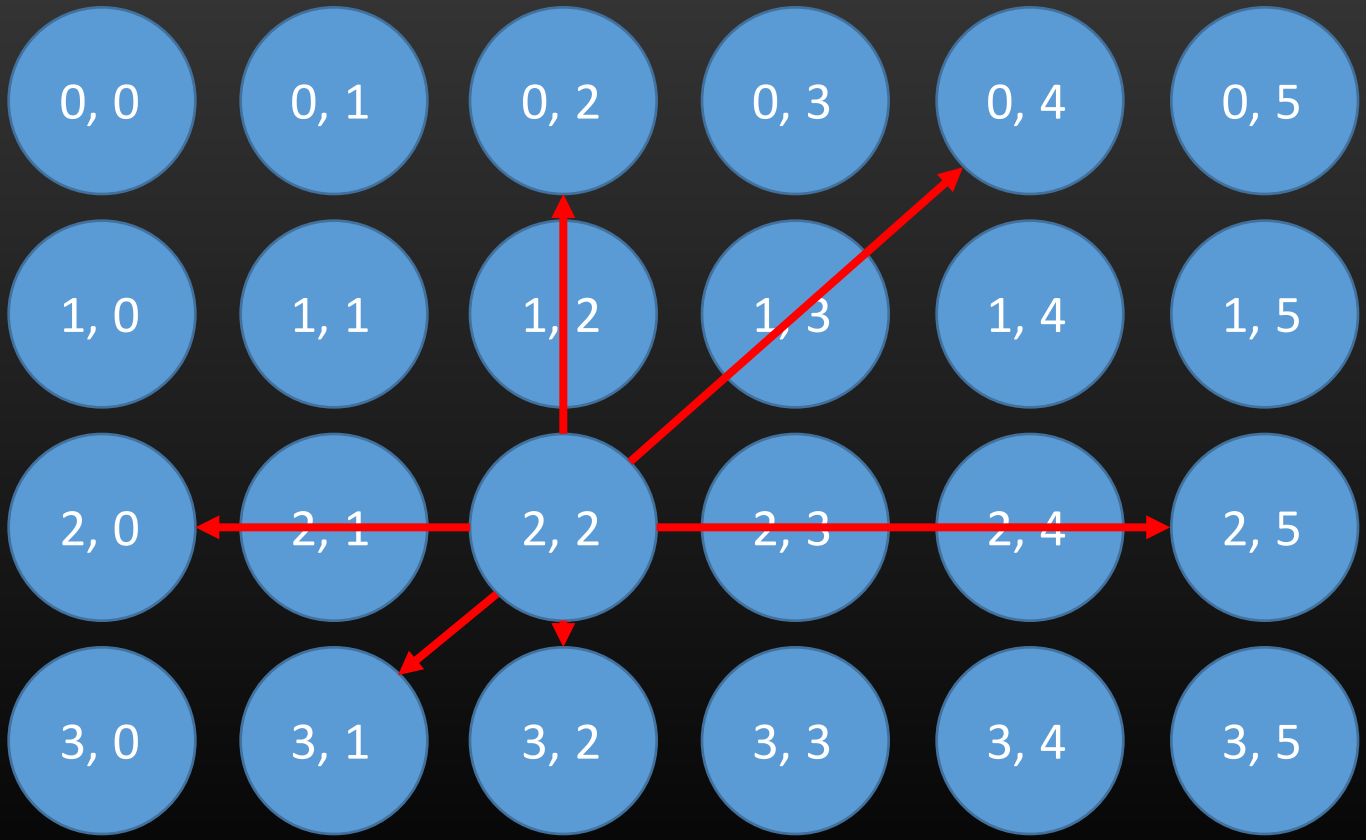
- The state (x, y) represents that there are x and y liter water in the first and second jug respectively
- The state (x, y) can transit to six states:
  - (x, 0) and (0, y) by emptying a jug
  - (N, y) and (x, M) by filling a jug completely
  - (x + y - M, M) or (0, x + y) by pouring water from the first jug to second
  - (N, x + y – N) or (x + y, 0) by pouring water from the second jug to first
- A graph can be built by using states as vertices and transitions as edges

States – Example 1

- For example, when N = 3, M = 5 and the required volume is 4:

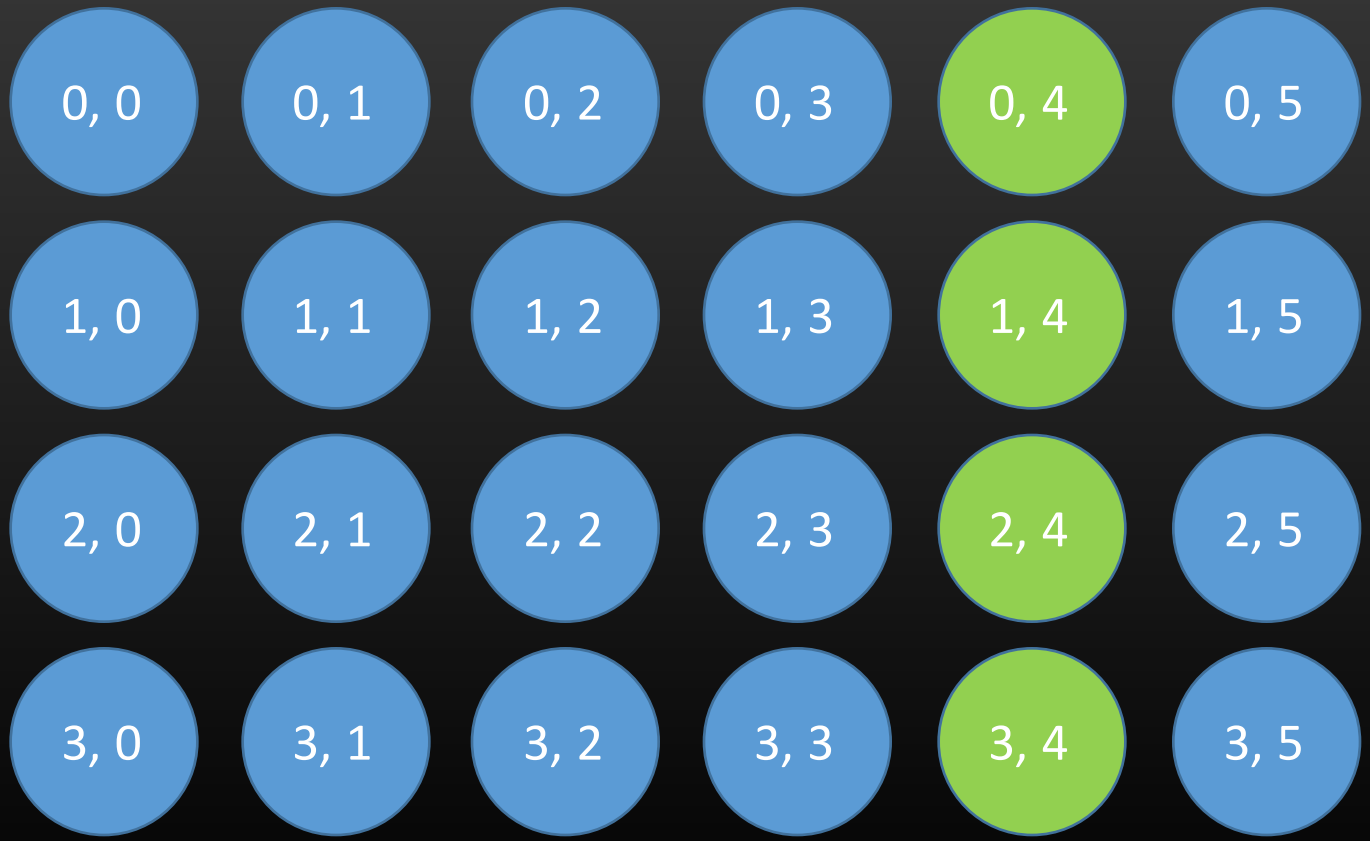


States – Example 1



States – Example 1

- The answer will be the shortest path from (0, 0) to these nodes



States – Example 2

- A rectangular maze
- You are only allowed to move one cell horizontally or vertically in one move
- Now you have 1 bomb to destroy a wall of exactly one cell
- Find the shortest path from one to another

States – Example 2

- The state (x, y, b) represents the you move to the cell (x, y) and still have b bombs
- (x, y, b) can transit to (x, y + 1, b) if the cell (x, y + 1) is not a wall, and (x, y, b) can transit to (x, y + 1, b) if the cell (x, y) is a wall and b is greater than zero
- Similar for other directions

States – Example 2